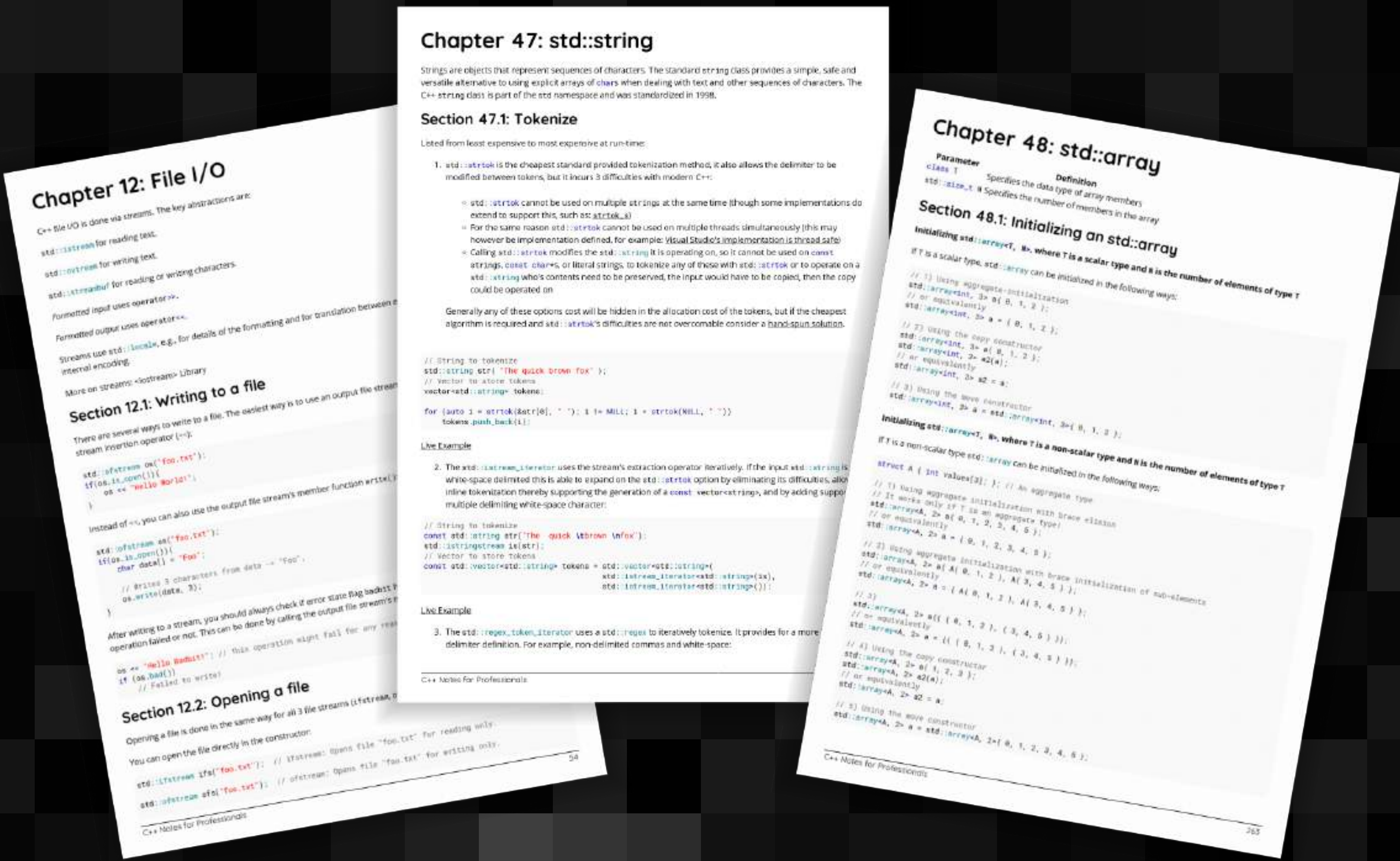
C++

Notes for Professionals



600+ pages

of professional hints and tricks

GoalKicker.com
Free Programming Books

Contents

About

Please feel free to share this PDF with anyone for free,
latest version of this book can be downloaded from:
https://goalkicker.com/CPlusPlusBook

This *C++ Notes for Professionals* book is compiled from Stack Overflow Documentation, the content is written by the beautiful people at Stack Overflow.

The information presented in this book is not guaranteed to be correct nor accurate, use at your own risk

Please send feedback and corrections to web@petercv.com

Chapter 1: Getting started with C++

Version	Standard	Release Date
C++98	ISO/IEC 14882:1998	1998-09-01
C++03	ISO/IEC 14882:2003	2003-10-16
C++11	ISO/IEC 14882:2011	2011-09-01
C++14	ISO/IEC 14882:2014	2014-12-15
C++17	TBD	2017-01-01
C++20	TBD	2020-01-01

Section 1.1: Hello World

This program prints `Hello World!` to the standard output stream:

```
#include <iostream>

int main()
{
    std::cout << "Hello World!" << std::endl;
}
```

See it live on Coliru.

Analysis

Let's examine each part of this code in detail:

- `#include <iostream>` is a **preprocessor directive** that includes the content of the standard C++ header file `iostream`.

  `iostream` is a **standard library header file** that contains definitions of the standard input and output streams. These definitions are included in the `std` namespace, explained below.

  The **standard input/output (I/O) streams** provide ways for programs to get input from and output to an external system -- usually the terminal.

- `int main() { ... }` defines a new function named `main`. By convention, the `main` function is called upon execution of the program. There must be only one `main` function in a C++ program, and it must always return a number of the `int` type.

  Here, the `int` is what is called the function's return type. The value returned by the `main` function is an **exit code.**

  By convention, a program exit code of `0` or `EXIT_SUCCESS` is interpreted as success by a system that executes the program. Any other return code is associated with an error.

  If no `return` statement is present, the `main` function (and thus, the program itself) returns `0` by default. In this example, we don't need to explicitly write `return 0;`.

  All other functions, except those that return the `void` type, must explicitly return a value according to their return type, or else must not return at all.

- `std::cout << "Hello World!" << std::endl;` prints "Hello World!" to the standard output stream:

    - `std` is a namespace, and `::` is the **scope resolution operator** that allows look-ups for objects by name within a namespace.

      There are many namespaces. Here, we use `::` to show we want to use `cout` from the `std` namespace. For more information refer to Scope Resolution Operator - Microsoft Documentation.

    - `std::cout` is the **standard output stream** object, defined in `iostream`, and it prints to the standard output (`stdout`).

    - `<<` is, *in this context*, the **stream insertion operator**, so called because it *inserts* an object into the *stream* object.

      The standard library defines the `<<` operator to perform data insertion for certain data types into output streams. `stream << content` inserts `content` into the stream and returns the same, but updated stream. This allows stream insertions to be chained: `std::cout << "Foo" << " Bar";` prints "FooBar" to the console.

    - `"Hello World!"` is a **character string literal**, or a "text literal." The stream insertion operator for character string literals is defined in file `iostream`.

    - `std::endl` is a special **I/O stream manipulator** object, also defined in file `iostream`. Inserting a manipulator into a stream changes the state of the stream.

      The stream manipulator `std::endl` does two things: first it inserts the end-of-line character and then it flushes the stream buffer to force the text to show up on the console. This ensures that the data inserted into the stream actually appear on your console. (Stream data is usually stored in a buffer and then "flushed" in batches unless you force a flush immediately.)

      An alternate method that avoids the flush is:

      ```
      std::cout << "Hello World!\n";
      ```

      where `\n` is the **character escape sequence** for the newline character.

    - The semicolon (`;`) notifies the compiler that a statement has ended. All C++ statements and class definitions require an ending/terminating semicolon.

Section 1.2: Comments

A **comment** is a way to put arbitrary text inside source code without having the C++ compiler interpret it with any functional meaning. Comments are used to give insight into the design or method of a program.

There are two types of comments in C++:

Single-Line Comments

The double forward-slash sequence `//` will mark all text until a newline as a comment:

```
int main()
{
```

```
    // This is a single-line comment.
    int a;  // this also is a single-line comment
    int i;  // this is another single-line comment
}
```

C-Style/Block Comments

The sequence `/*` is used to declare the start of the comment block and the sequence `*/` is used to declare the end of comment. All text between the start and end sequences is interpreted as a comment, even if the text is otherwise valid C++ syntax. These are sometimes called "C-style" comments, as this comment syntax is inherited from C++'s predecessor language, C:

```
int main()
{
    /*
     *  This is a block comment.
     */
    int a;
}
```

In any block comment, you can write anything you want. When the compiler encounters the symbol `*/`, it terminates the block comment:

```
int main()
{
    /* A block comment with the symbol /*
       Note that the compiler is not affected by the second /*
       however, once the end-block-comment symbol is reached,
       the comment ends.
    */
    int a;
}
```

The above example is valid C++ (and C) code. However, having additional `/*` inside a block comment might result in a warning on some compilers.

Block comments can also start and end *within* a single line. For example:

```
void SomeFunction(/* argument 1 */ int a, /* argument 2 */ int b);
```

Importance of Comments

As with all programming languages, comments provide several benefits:

- Explicit documentation of code to make it easier to read/maintain
- Explanation of the purpose and functionality of code
- Details on the history or reasoning behind the code
- Placement of copyright/licenses, project notes, special thanks, contributor credits, etc. directly in the source code.

However, comments also have their downsides:

- They must be maintained to reflect any changes in the code
- Excessive comments tend to make the code *less* readable

The need for comments can be reduced by writing clear, self-documenting code. A simple example is the use of explanatory names for variables, functions, and types. Factoring out logically related tasks into discrete functions goes hand-in-hand with this.

Comment markers used to disable code

During development, comments can also be used to quickly disable portions of code without deleting it. This is often useful for testing or debugging purposes, but is not good style for anything other than temporary edits. This is often referred to as "commenting out".

Similarly, keeping old versions of a piece of code in a comment for reference purposes is frowned upon, as it clutters files while offering little value compared to exploring the code's history via a versioning system.

Section 1.3: The standard C++ compilation process

Executable C++ program code is usually produced by a compiler.

A **compiler** is a program that translates code from a programming language into another form which is (more) directly executable for a computer. Using a compiler to translate code is called **compilation.**

C++ inherits the form of its compilation process from its "parent" language, C. Below is a list showing the four major steps of compilation in C++:

1. The C++ preprocessor copies the contents of any included header files into the source code file, generates macro code, and replaces symbolic constants defined using #define with their values.
2. The expanded source code file produced by the C++ preprocessor is compiled into assembly language appropriate for the platform.
3. The assembler code generated by the compiler is assembled into appropriate object code for the platform.
4. The object code file generated by the assembler is linked together with the object code files for any library functions used to produce an executable file.

- Note: some compiled code is linked together, but not to create a final program. Usually, this "linked" code can also be packaged into a format that can be used by other programs. This "bundle of packaged, usable code" is what C++ programmers refer to as a **library.**

Many C++ compilers may also merge or un-merge certain parts of the compilation process for ease or for additional analysis. Many C++ programmers will use different tools, but all of the tools will generally follow this generalized process when they are involved in the production of a program.

The link below extends this discussion and provides a nice graphic to help. [1]:
http://faculty.cs.niu.edu/~mcmahon/CS241/Notes/compile.html

Section 1.4: Function

A **function** is a unit of code that represents a sequence of statements.

Functions can accept **arguments** or values and **return** a single value (or not). To use a function, a **function call** is used on argument values and the use of the function call itself is replaced with its return value.

Every function has a **type signature** -- the types of its arguments and the type of its return type.

Functions are inspired by the concepts of the procedure and the mathematical function.

- Note: C++ functions are essentially procedures and do not follow the exact definition or rules of mathematical functions.

Functions are often meant to perform a specific task. and can be called from other parts of a program. A function must be declared and defined before it is called elsewhere in a program.

- Note: popular function definitions may be hidden in other included files (often for convenience and reuse across many files). This is a common use of header files.

Function Declaration

A **function declaration** is declares the existence of a function with its name and type signature to the compiler. The syntax is as the following:

```
int add2(int i); // The function is of the type (int) -> (int)
```

In the example above, the `int add2(int i)` function declares the following to the compiler:

- The **return type** is `int`.
- The **name** of the function is `add2`.
- The **number of arguments** to the function is 1:
  - The first argument is of the type `int`.
  - The first argument will be referred to in the function's contents by the name `i`.

The argument name is optional; the declaration for the function could also be the following:

```
int add2(int); // Omitting the function arguments' name is also permitted.
```

Per the **one-definition rule**, a function with a certain type signature can only be declared or defined once in an entire C++ code base visible to the C++ compiler. In other words, functions with a specific type signature cannot be re-defined -- they must only be defined once. Thus, the following is not valid C++:

```
int add2(int i);  // The compiler will note that add2 is a function (int) -> int
int add2(int j);  // As add2 already has a definition of (int) -> int, the compiler
                  // will regard this as an error.
```

If a function returns nothing, its return type is written as `void`. If it takes no parameters, the parameter list should be empty.

```
void do_something(); // The function takes no parameters, and does not return anything.
                     // Note that it can still affect variables it has access to.
```

Function Call

A function can be called after it has been declared. For example, the following program calls `add2` with the value of `2` within the function of `main`:

```
#include <iostream>

int add2(int i);    // Declaration of add2

// Note: add2 is still missing a DEFINITION.
// Even though it doesn't appear directly in code,
// add2's definition may be LINKED in from another object file.

int main()
{
    std::cout << add2(2) << "\n";  // add2(2) will be evaluated at this point,
                                   // and the result is printed.
    return 0;
}
```

Here, `add2(2)` is the syntax for a function call.

Function Definition

A *function definition** is similar to a declaration, except it also contains the code that is executed when the function is called within its body.

An example of a function definition for `add2` might be:

```
int add2(int i)       // Data that is passed into (int i) will be referred to by the name i
{                     // while in the function's curly brackets or "scope."

    int j = i + 2;    // Definition of a variable j as the value of i+2.
    return j;         // Returning or, in essence, substitution of j for a function call to
                      // add2.
}
```

Function Overloading

You can create multiple functions with the same name but different parameters.

```
int add2(int i)           // Code contained in this definition will be evaluated
{                         // when add2() is called with one parameter.
    int j = i + 2;
    return j;
}

int add2(int i, int j)    // However, when add2() is called with two parameters, the
{                         // code from the initial declaration will be overloaded,
    int k = i + j + 2 ;   // and the code in this declaration will be evaluated
    return k;             // instead.
}
```

Both functions are called by the same name `add2`, but the actual function that is called depends directly on the amount and type of the parameters in the call. In most cases, the C++ compiler can compute which function to call. In some cases, the type must be explicitly stated.

Default Parameters

Default values for function parameters can only be specified in function declarations.

```
int multiply(int a, int b = 7); // b has default value of 7.
int multiply(int a, int b)
{
    return a * b;               // If multiply() is called with one parameter, the
}                               // value will be multiplied by the default, 7.
```

In this example, `multiply()` can be called with one or two parameters. If only one parameter is given, `b` will have default value of 7. Default arguments must be placed in the latter arguments of the function. For example:

```
int multiply(int a = 10, int b = 20); // This is legal
int multiply(int a = 10, int b);      // This is illegal since int a is in the former
```

Special Function Calls - Operators

There exist special function calls in C++ which have different syntax than `name_of_function(value1, value2, value3)`. The most common example is that of operators.

Certain special character sequences that will be reduced to function calls by the compiler, such as `!`, `+`, `-`, `*`, `%`, and `<<` and many more. These special characters are normally associated with non-programming usage or are used for

aesthetics (e.g. the + character is commonly recognized as the addition symbol both within C++ programming as well as in elementary math).

C++ handles these character sequences with a special syntax; but, in essence, each occurrence of an operator is reduced to a function call. For example, the following C++ expression:

```
3+3
```

is equivalent to the following function call:

```
operator+(3, 3)
```

All operator function names start with `operator`.

While in C++'s immediate predecessor, C, operator function names cannot be assigned different meanings by providing additional definitions with different type signatures, in C++, this is valid. "Hiding" additional function definitions under one unique function name is referred to as **operator overloading** in C++, and is a relatively common, but not universal, convention in C++.

Section 1.5: Visibility of function prototypes and declarations

In C++, code must be declared or defined before usage. For example, the following produces a compile time error:

```
int main()
{
  foo(2); // error: foo is called, but has not yet been declared
}

void foo(int x) // this later definition is not known in main
{
}
```

There are two ways to resolve this: putting either the definition or declaration of `foo()` before its usage in `main()`. Here is one example:

```
void foo(int x) {}  //Declare the foo function and body first

int main()
{
  foo(2); // OK: foo is completely defined beforehand, so it can be called here.
}
```

However it is also possible to "forward-declare" the function by putting only a "prototype" declaration before its usage and then defining the function body later:

```
void foo(int);  // Prototype declaration of foo, seen by main
                // Must specify return type, name, and argument list types
int main()
{
  foo(2); // OK: foo is known, called even though its body is not yet defined
}

void foo(int x) //Must match the prototype
{
    // Define body of foo here
}
```

The prototype must specify the return type (`void`), the name of the function (`foo`), and the argument list variable types (`int`), but the names of the arguments are NOT required.

One common way to integrate this into the organization of source files is to make a header file containing all of the prototype declarations:

```
// foo.h
void foo(int); // prototype declaration
```

and then provide the full definition elsewhere:

```
// foo.cpp --> foo.o
#include "foo.h" // foo's prototype declaration is "hidden" in here
void foo(int x) { } // foo's body definition
```

and then, once compiled, link the corresponding object file `foo.o` into the compiled object file where it is used in the linking phase, `main.o`:

```
// main.cpp --> main.o
#include "foo.h" // foo's prototype declaration is "hidden" in here
int main() { foo(2); } // foo is valid to call because its prototype declaration was beforehand.
// the prototype and body definitions of foo are linked through the object files
```

An "unresolved external symbol" error occurs when the function *prototype* and *call* exist, but the function *body* is not defined. These can be trickier to resolve as the compiler won't report the error until the final linking stage, and it doesn't know which line to jump to in the code to show the error.

Section 1.6: Preprocessor

The preprocessor is an important part of the compiler.

It edits the source code, cutting some bits out, changing others, and adding other things.

In source files, we can include preprocessor directives. These directives tells the preprocessor to perform specific actions. A directive starts with a # on a new line. Example:

```
#define ZERO 0
```

The first preprocessor directive you will meet is probably the

```
#include <something>
```

directive. What it does is takes all of `something` and inserts it in your file where the directive was. The hello world program starts with the line

```
#include <iostream>
```

This line adds the functions and objects that let you use the standard input and output.

The C language, which also uses the preprocessor, does not have as many header files as the C++ language, but in C++ you can use all the C header files.

The next important directive is probably the

```
#define something something_else
```

directive. This tells the preprocessor that as it goes along the file, it should replace every occurrence of `something` with `something_else`. It can also make things similar to functions, but that probably counts as advanced C++.

The `something_else` is not needed, but if you define `something` as nothing, then outside preprocessor directives, all occurrences of `something` will vanish.

This actually is useful, because of the `#if`,`#else` and `#ifdef` directives. The format for these would be the following:

```
#if something==true
//code
#else
//more code
#endif

#ifdef thing_that_you_want_to_know_if_is_defined
//code
#endif
```

These directives insert the code that is in the true bit, and deletes the false bits. this can be used to have bits of code that are only included on certain operating systems, without having to rewrite the whole code.

Chapter 2: Literals

Traditionally, a literal is an expression denoting a constant whose type and value are evident from its spelling. For example, `42` is a literal, while `x` is not since one must see its declaration to know its type and read previous lines of code to know its value.

However, C++11 also added user-defined literals, which are not literals in the traditional sense but can be used as a shorthand for function calls.

Section 2.1: this

Within a member function of a class, the keyword `this` is a pointer to the instance of the class on which the function was called. `this` cannot be used in a static member function.

```
struct S {
    int x;
    S& operator=(const S& other) {
        x = other.x;
        // return a reference to the object being assigned to
        return *this;
    }
};
```

The type of `this` depends on the cv-qualification of the member function: if `X::f` is `const`, then the type of `this` within `f` is `const X*`, so `this` cannot be used to modify non-static data members from within a `const` member function. Likewise, `this` inherits `volatile` qualification from the function it appears in.

Version ≥ C++11

`this` can also be used in a *brace-or-equal-initializer* for a non-static data member.

```
struct S;
struct T {
    T(const S* s);
    // ...
};
struct S {
    // ...
    T t{this};
};
```

`this` is an rvalue, so it cannot be assigned to.

Section 2.2: Integer literal

An integer literal is a primary expression of the form

- decimal-literal

It is a non-zero decimal digit (1, 2, 3, 4, 5, 6, 7, 8, 9), followed by zero or more decimal digits (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)

```
int d = 42;
```

- octal-literal

It is the digit zero (0) followed by zero or more octal digits (0, 1, 2, 3, 4, 5, 6, 7)

```
int o = 052
```

- hex-literal

It is the character sequence 0x or the character sequence 0X followed by one or more hexadecimal digits (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, A, b, B, c, C, d, D, e, E, f, F)

```
int x = 0x2a; int X = 0X2A;
```

- binary-literal (since C++14)

It is the character sequence 0b or the character sequence 0B followed by one or more binary digits (0, 1)

```
int b = 0b101010; // C++14
```

Integer-suffix, if provided, may contain one or both of the following (if both are provided, they may appear in any order:

- unsigned-suffix (the character u or the character U)

```
unsigned int u_1 = 42u;
```

- long-suffix (the character l or the character L) or the long-long-suffix (the character sequence ll or the character sequence LL) (since C++11)

The following variables are also initialized to the same value:

```
unsigned long long l1 = 18446744073709550592ull; // C++11
unsigned long long l2 = 18'446'744'073'709'550'592llu; // C++14
unsigned long long l3 = 1844'6744'0737'0955'0592uLL; // C++14
unsigned long long l4 = 184467'440737'0'95505'92LLU; // C++14
```

Notes

Letters in the integer literals are case-insensitive: 0xDeAdBaBeU and 0XdeadBABEu represent the same number (one exception is the long-long-suffix, which is either ll or LL, never lL or Ll)

There are no negative integer literals. Expressions such as -1 apply the unary minus operator to the value represented by the literal, which may involve implicit type conversions.

In C prior to C99 (but not in C++), unsuffixed decimal values that do not fit in long int are allowed to have the type unsigned long int.

When used in a controlling expression of #if or #elif, all signed integer constants act as if they have type std::intmax_t and all unsigned integer constants act as if they have type std::uintmax_t.

Section 2.3: true

A keyword denoting one of the two possible values of type `bool`.

```
bool ok = true;
if (!f()) {
    ok = false;
    goto end;
}
```

Section 2.4: false

A keyword denoting one of the two possible values of type `bool`.

```
bool ok = true;
if (!f()) {
    ok = false;
    goto end;
}
```

Section 2.5: nullptr

Version ≥ C++11

A keyword denoting a null pointer constant. It can be converted to any pointer or pointer-to-member type, yielding a null pointer of the resulting type.

```
Widget* p = new Widget();
delete p;
p = nullptr; // set the pointer to null after deletion
```

Note that `nullptr` is not itself a pointer. The type of `nullptr` is a fundamental type known as `std::nullptr_t`.

```
void f(int* p);

template <class T>
void g(T* p);

void h(std::nullptr_t p);

int main() {
    f(nullptr); // ok
    g(nullptr); // error
    h(nullptr); // ok
}
```

Chapter 3: operator precedence

Section 3.1: Logical && and || operators: short-circuit

&& has precedence over ||, this means that parentheses are placed to evaluate what would be evaluated together.

c++ uses short-circuit evaluation in && and || to not do unnecessary executions.
If the left hand side of || returns true the right hand side does not need to be evaluated anymore.

```
#include <iostream>
#include <string>

using namespace std;

bool True(string id){
    cout << "True" << id << endl;
    return true;
}

bool False(string id){
    cout << "False" << id << endl;
    return false;
}


int main(){
    bool result;
    //let's evaluate 3 booleans with || and && to illustrate operator precedence
    //precedence does not mean that && will be evaluated first but rather where
    //parentheses would be added
    //example 1
    result =
        False("A") || False("B") && False("C");
                // eq. False("A") || (False("B") && False("C"))
    //FalseA
    //FalseB
    //"Short-circuit evaluation skip of C"
    //A is false so we have to evaluate the right of ||,
    //B being false we do not have to evaluate C to know that the result is false



    result =
        True("A") || False("B") && False("C");
                // eq. True("A") || (False("B") && False("C"))
    cout << result << " :=====================" << endl;
    //TrueA
    //"Short-circuit evaluation skip of B"
    //"Short-circuit evaluation skip of C"
    //A is true so we do not have to evaluate
    //        the right of || to know that the result is true
    //If || had precedence over && the equivalent evaluation would be:
    // (True("A") || False("B")) && False("C")
    //What would print
    //TrueA
    //"Short-circuit evaluation skip of B"
    //FalseC
    //Because the parentheses are placed differently
    //the parts that get evaluated are differently
    //which makes that the end result in this case would be False because C is false
```

```
}
```

Section 3.2: Unary Operators

Unary operators act on the object upon which they are called and have high precedence. (See Remarks)

When used postfix, the action occurs only after the entire operation is evaluated, leading to some interesting arithmetics:

```
int a = 1;
++a;            // result: 2
a--;            // result: 1
int minusa=-a;  // result: -1

bool b = true;
!b; // result: true

a=4;
int c = a++/2;      // equal to: (a==4) 4 / 2   result: 2 ('a' incremented postfix)
cout << a << endl;  // prints 5!
int d = ++a/2;      // equal to: (a+1) == 6 / 2 result: 3

int arr[4] =  {1,2,3,4};

int *ptr1 = &arr[0];    // points to arr[0] which is 1
int *ptr2 = ptr1++;     // ptr2 points to arr[0] which is still 1; ptr1 incremented
std::cout << *ptr1++ << std::endl;  // prints  2

int e = arr[0]++;       // receives the value of arr[0] before it is incremented
std::cout << e << std::endl;      // prints 1
std::cout << *ptr2 << std::endl;  // prints arr[0] which is now 2
```

Section 3.3: Arithmetic operators

Arithmetic operators in C++ have the same precedence as they do in mathematics:

Multiplication and division have left associativity(meaning that they will be evaluated from left to right) and they have higher precedence than addition and subtraction, which also have left associativity.

We can also force the precedence of expression using parentheses `( )`. Just the same way as you would do that in normal mathematics.

```
// volume of a spherical shell = 4 pi R^3 - 4 pi r^3
double vol = 4.0*pi*R*R*R/3.0 - 4.0*pi*r*r*r/3.0;

//Addition:

int a = 2+4/2;          // equal to: 2+(4/2)         result: 4
int b = (3+3)/2;        // equal to: (3+3)/2         result: 3

//With Multiplication

int c = 3+4/2*6;        // equal to: 3+((4/2)*6)     result: 15
int d = 3*(3+6)/9;      // equal to: (3*(3+6))/9     result: 3

//Division and Modulo

int g = 3-3%1;          // equal to: 3 % 1 = 0  3 - 0 = 3
int h = 3-(3%1);        // equal to: 3 % 1 = 0  3 - 0 = 3
```

```
int i = 3-3/1%3;        // equal to: 3 / 1 = 3  3 % 3 = 0  3 - 0 = 3
int l = 3-(3/1)%3;      // equal to: 3 / 1 = 3  3 % 3 = 0  3 - 0 = 3
int m = 3-(3/(1%3));    // equal to: 1 % 3 = 1  3 / 1 = 3  3 - 3 = 0
```

Section 3.4: Logical AND and OR operators

These operators have the usual precedence in C++: AND before OR.

```
// You can drive with a foreign license for up to 60 days
bool can_drive = has_domestic_license || has_foreign_license && num_days <= 60;
```

This code is equivalent to the following:

```
// You can drive with a foreign license for up to 60 days
bool can_drive = has_domestic_license || (has_foreign_license && num_days <= 60);
```

Adding the parenthesis does not change the behavior, though, it does make it easier to read. By adding these parentheses, no confusion exist about the intent of the writer.

Chapter 4: Floating Point Arithmetic

Section 4.1: Floating Point Numbers are Weird

The first mistake that nearly every single programmer makes is presuming that this code will work as intended:

```
float total = 0;
for(float a = 0; a != 2; a += 0.01f) {
    total += a;
}
```

The novice programmer assumes that this will sum up every single number in the range `0, 0.01, 0.02, 0.03, ..., 1.97, 1.98, 1.99`, to yield the result `199`—the mathematically correct answer.

Two things happen that make this untrue:

1. The program as written never concludes. `a` never becomes equal to `2`, and the loop never terminates.
2. If we rewrite the loop logic to check `a < 2` instead, the loop terminates, but the total ends up being something different from `199`. On IEEE754-compliant machines, it will often sum up to about `201` instead.

The reason that this happens is that **Floating Point Numbers represent Approximations of their assigned values**.

The classical example is the following computation:

```
double a = 0.1;
double b = 0.2;
double c = 0.3;
if(a + b == c)
    //This never prints on IEEE754-compliant machines
    std::cout << "This Computer is Magic!" << std::endl;
else
    std::cout << "This Computer is pretty normal, all things considered." << std::endl;
```

Though what we the programmer see is three numbers written in base10, what the compiler (and the underlying hardware) see are binary numbers. Because `0.1`, `0.2`, and `0.3` require perfect division by `10`—which is quite easy in a base-10 system, but impossible in a base-2 system—these numbers have to be stored in imprecise formats, similar to how the number `1/3` has to be stored in the imprecise form `0.333333333333333...` in base-10.

```
//64-bit floats have 53 digits of precision, including the whole-number-part.
double a =     0011111110111001100110011001100110011001100110011001100110011010; //imperfect
representation of 0.1
double b =     0011111111001001100110011001100110011001100110011001100110011010; //imperfect
representation of 0.2
double c =     0011111111010011001100110011001100110011001100110011001100110011; //imperfect
representation of 0.3
double a + b = 0011111111010011001100110011001100110011001100110011001100110100; //Note that this
is not quite equal to the "canonical" 0.3!
```

Chapter 5: Bit Operators

Section 5.1: | - bitwise OR

```
int a = 5;     // 0101b  (0x05)
int b = 12;    // 1100b  (0x0C)
int c = a | b; // 1101b  (0x0D)

std::cout << "a = " << a << ", b = " << b << ", c = " << c << std::endl;
```

Output

```
a = 5, b = 12, c = 13
```

Why

A bit wise `OR` operates on the bit level and uses the following Boolean truth table:

```
true OR true = true
true OR false = true
false OR false = false
```

When the binary value for `a` (`0101`) and the binary value for `b` (`1100`) are `OR`'ed together we get the binary value of `1101`:

```
int a = 0 1 0 1
int b = 1 1 0 0 |
        ---------
int c = 1 1 0 1
```

The bit wise OR does not change the value of the original values unless specifically assigned to using the bit wise assignment compound operator `|=`:

```
int a = 5;  // 0101b  (0x05)
a |= 12;    // a = 0101b | 1101b
```

Section 5.2: ^ - bitwise XOR (exclusive OR)

```
int a = 5;     // 0101b  (0x05)
int b = 9;     // 1001b  (0x09)
int c = a ^ b; // 1100b  (0x0C)

std::cout << "a = " << a << ", b = " << b << ", c = " << c << std::endl;
```

Output

```
a = 5, b = 9, c = 12
```

Why

A bit wise `XOR` (exclusive or) operates on the bit level and uses the following Boolean truth table:

```
true OR true = false
true OR false = true
false OR false = false
```

Notice that with an XOR operation `true OR true = false` where as with operations `true AND/OR true = true`, hence the exclusive nature of the XOR operation.

Using this, when the binary value for `a` (`0101`) and the binary value for `b` (`1001`) are `XOR`'ed together we get the binary value of `1100`:

```
int a = 0 1 0 1
int b = 1 0 0 1 ^
        ---------
int c = 1 1 0 0
```

The bit wise XOR does not change the value of the original values unless specifically assigned to using the bit wise assignment compound operator `^=`:

```
int a = 5;  // 0101b  (0x05)
a ^= 9;    // a = 0101b ^ 1001b
```

The bit wise XOR can be utilized in many ways and is often utilized in bit mask operations for encryption and compression.

Note: The following example is often shown as an example of a nice trick. But should not be used in production code (there are better ways `std::swap()` to achieve the same result).

You can also utilize an XOR operation to swap two variables without a temporary:

```
int a = 42;
int b = 64;

// XOR swap
a ^= b;
b ^= a;
a ^= b;

std::cout << "a = " << a << ", b = " << b << "\n";
```

To productionalize this you need to add a check to make sure it can be used.

```
void doXORSwap(int& a, int& b)
{
    // Need to add a check to make sure you are not swapping the same
    // variable with itself. Otherwise it will zero the value.
    if (&a != &b)
    {
        // XOR swap
        a ^= b;
        b ^= a;
        a ^= b;
    }
}
```

So though it looks like a nice trick in isolation it is not useful in real code. xor is not a base logical operation,but a combination of others: a^c=~(a&c)&(a|c)

also in 2015+ compilers variables may be assigned as binary:

```
int cn=0b0111;
```

Section 5.3: & - bitwise AND

```
int a = 6;     // 0110b  (0x06)
int b = 10;    // 1010b  (0x0A)
int c = a & b; // 0010b  (0x02)

std::cout << "a = " << a << ", b = " << b << ", c = " << c << std::endl;
```

Output

`a = 6, b = 10, c = 2`

Why

A bit wise `AND` operates on the bit level and uses the following Boolean truth table:

```
TRUE  AND TRUE  = TRUE
TRUE  AND FALSE = FALSE
FALSE AND FALSE = FALSE
```

When the binary value for `a` (`0110`) and the binary value for `b` (`1010`) are `AND`'ed together we get the binary value of `0010`:

```
int a = 0 1 1 0
int b = 1 0 1 0 &
        ---------
int c = 0 0 1 0
```

The bit wise AND does not change the value of the original values unless specifically assigned to using the bit wise assignment compound operator `&=`:

```
int a = 5;  // 0101b  (0x05)
a &= 10;    // a = 0101b & 1010b
```

Section 5.4: << - left shift

```
int a = 1;      // 0001b
int b = a << 1; // 0010b

std::cout << "a = " << a << ", b = " << b << std::endl;
```

Output

`a = 1, b = 2`

Why

The left bit wise shift will shift the bits of the left hand value (`a`) the number specified on the right (`1`), essentially padding the least significant bits with 0's, so shifting the value of `5` (binary `0000 0101`) to the left 4 times (e.g. `5 << 4`) will yield the value of `80` (binary `0101 0000`). You might note that shifting a value to the left 1 time is also the same as multiplying the value by 2, example:

```
int a = 7;
while (a < 200) {
    std::cout << "a = " << a << std::endl;
    a <<= 1;
```

```
}

a = 7;
while (a < 200) {
    std::cout << "a = " << a << std::endl;
    a *= 2;
}
```

But it should be noted that the left shift operation will shift *all* bits to the left, including the sign bit, example:

```
int a = 2147483647; // 0111 1111 1111 1111 1111 1111 1111 1111
int b = a << 1;     // 1111 1111 1111 1111 1111 1111 1111 1110

std::cout << "a = " << a << ", b = " << b << std::endl;
```

Possible output: `a = 2147483647, b = -2`

While some compilers will yield results that seem expected, it should be noted that if you left shift a signed number so that the sign bit is affected, the result is **undefined**. It is also **undefined** if the number of bits you wish to shift by is a negative number or is larger than the number of bits the type on the left can hold, example:

```
int a = 1;
int b = a << -1;  // undefined behavior
char c = a << 20; // undefined behavior
```

The bit wise left shift does not change the value of the original values unless specifically assigned to using the bit wise assignment compound operator `<<=`:

```
int a = 5;  // 0101b
a <<= 1;    // a = a << 1;
```

Section 5.5: >> - right shift

```
int a = 2;      // 0010b
int b = a >> 1; // 0001b

std::cout << "a = " << a << ", b = " << b << std::endl;
```

Output

`a = 2, b = 1`

Why

The right bit wise shift will shift the bits of the left hand value (`a`) the number specified on the right (`1`); it should be noted that while the operation of a right shift is standard, what happens to the bits of a right shift on a *signed negative* number is *implementation defined* and thus cannot be guaranteed to be portable, example:

```
int a = -2;
int b = a >> 1; // the value of b will be depend on the compiler
```

It is also undefined if the number of bits you wish to shift by is a negative number, example:

```
int a = 1;
int b = a >> -1;  // undefined behavior
```

The bit wise right shift does not change the value of the original values unless specifically assigned to using the bit wise assignment compound operator `>>=`:

```
int a = 2;  // 0010b
a >>= 1;    // a = a >> 1;
```

Chapter 6: Bit Manipulation

Section 6.1: Remove rightmost set bit

C-style bit-manipulation

```
template <typename T>
T rightmostSetBitRemoved(T n)
{
    // static_assert(std::is_integral<T>::value && !std::is_signed<T>::value, "type should be
unsigned"); // For c++11 and later
    return n & (n - 1);
}
```

Explanation

- if `n` is zero, we have `0 & 0xFF..FF` which is zero
- else `n` can be written `0bxxxxxx10..00` and `n - 1` is `0bxxxxxx011..11`, so `n & (n - 1)` is `0bxxxxxx000..00`.

Section 6.2: Set all bits

C-style bit-manipulation

```
x = -1; // -1 == 1111 1111 ... 1111b
```

(See here for an explanation of why this works and is actually the best approach.)

Using std::bitset

```
std::bitset<10> x;
x.set(); // Sets all bits to '1'
```

Section 6.3: Toggling a bit

C-style bit-manipulation

A bit can be toggled using the XOR operator (`^`).

```
// Bit x will be the opposite value of what it is currently
number ^= 1LL << x;
```

Using std::bitset

```
std::bitset<4> num(std::string("0100"));
num.flip(2); // num is now 0000
num.flip(0); // num is now 0001
num.flip();  // num is now 1110 (flips all bits)
```

Section 6.4: Checking a bit

C-style bit-manipulation

The value of the bit can be obtained by shifting the number to the right `x` times and then performing bitwise AND (`&`) on it:

```
(number >> x) & 1LL;  // 1 if the 'x'th bit of 'number' is set, 0 otherwise
```

The right-shift operation may be implemented as either an arithmetic (signed) shift or a logical (unsigned) shift. If

`number` in the expression `number >> x` has a signed type and a negative value, the resulting value is implementation-defined.

If we need the value of that bit directly in-place, we could instead left shift the mask:

```
(number & (1LL << x));  // (1 << x) if the 'x'th bit of 'number' is set, 0 otherwise
```

Either can be used as a conditional, since all non-zero values are considered true.

Using std::bitset

```
std::bitset<4> num(std::string("0010"));
bool bit_val = num.test(1);  // bit_val value is set to true;
```

Section 6.5: Counting bits set

The population count of a bitstring is often needed in cryptography and other applications and the problem has been widely studied.

The naive way requires one iteration per bit:

```
unsigned value = 1234;
unsigned bits = 0;  // accumulates the total number of bits set in `n`

for (bits = 0; value; value >>= 1)
  bits += value & 1;
```

A nice trick (based on Remove rightmost set bit) is:

```
unsigned bits = 0;  // accumulates the total number of bits set in `n`

for (; value; ++bits)
  value &= value - 1;
```

It goes through as many iterations as there are set bits, so it's good when `value` is expected to have few nonzero bits.

The method was first proposed by Peter Wegner (in CACM 3 / 322 - 1960) and it's well known since it appears in *C Programming Language* by Brian W. Kernighan and Dennis M. Ritchie.

This requires 12 arithmetic operations, one of which is a multication:

```
unsigned popcount(std::uint64_t x)
{
  const std::uint64_t m1  = 0x5555555555555555;  // binary: 0101...
  const std::uint64_t m2  = 0x3333333333333333;  // binary: 00110011..
  const std::uint64_t m4  = 0x0f0f0f0f0f0f0f0f;  // binary: 0000111100001111

  x -= (x >> 1) & m1;             // put count of each 2 bits into those 2 bits
  x = (x & m2) + ((x >> 2) & m2); // put count of each 4 bits into those 4 bits
  x = (x + (x >> 4)) & m4;        // put count of each 8 bits into those 8 bits
  return (x * h01) >> 56;  // left 8 bits of x + (x<<8) + (x<<16) + (x<<24) + ...
}
```

This kind of implementation has the best worst-case behavior (see Hamming weight for further details).

Many CPUs have a specific instruction (like x86's `popcnt`) and the compiler could offer a specific (**non standard**)

built in function. E.g. with g++ there is:

```
int __builtin_popcount (unsigned x);
```

Section 6.6: Check if an integer is a power of 2

The `n & (n - 1)` trick (see Remove rightmost set bit) is also useful to determine if an integer is a power of 2:

```
bool power_of_2 = n && !(n & (n - 1));
```

Note that without the first part of the check (`n &&`), `0` is incorrectly considered a power of 2.

Section 6.7: Setting a bit

C-style bit manipulation

A bit can be set using the bitwise OR operator (`|`).

```
// Bit x will be set
number |= 1LL << x;
```

Using std::bitset

`set(x)` or `set(x,true)` - sets bit at position `x` to `1`.

```
std::bitset<5> num(std::string("01100"));
num.set(0);      // num is now 01101
num.set(2);      // num is still 01101
num.set(4,true); // num is now 11110
```

Section 6.8: Clearing a bit

C-style bit-manipulation

A bit can be cleared using the bitwise AND operator (`&`).

```
// Bit x will be cleared
number &= ~(1LL << x);
```

Using std::bitset

`reset(x)` or `set(x,false)` - clears the bit at position `x`.

```
std::bitset<5> num(std::string("01100"));
num.reset(2);     // num is now 01000
num.reset(0);     // num is still 01000
num.set(3,false); // num is now 00000
```

Section 6.9: Changing the nth bit to x

C-style bit-manipulation

```
// Bit n will be set if x is 1 and cleared if x is 0.
number ^= (-x ^ number) & (1LL << n);
```

Using std::bitset

`set(n,val)` - sets bit `n` to the value `val`.

```
std::bitset<5> num(std::string("00100"));
num.set(0,true);  // num is now 00101
num.set(2,false); // num is now 00001
```

Section 6.10: Bit Manipulation Application: Small to Capital Letter

One of several applications of bit manipulation is converting a letter from small to capital or vice versa by choosing a **mask** and a proper **bit operation**. For example, the **a** letter has this binary representation `01(1)00001` while its capital counterpart has `01(0)00001`. They differ solely in the bit in parenthesis. In this case, converting the **a** letter from small to capital is basically setting the bit in parenthesis to one. To do so, we do the following:

```
/****************************************
convert small letter to captial letter.
========================================
     a: 01100001
  mask: 11011111  <-- (0xDF)  11(0)11111
      :---------
a&mask: 01000001  <-- A letter
*****************************************/
```

The code for converting a letter to A letter is

```
#include <cstdio>

int main()
{
    char op1 = 'a';  // "a" letter (i.e. small case)
    int mask = 0xDF; // choosing a proper mask

    printf("a (AND) mask = A\n");
    printf("%c   &   0xDF = %c\n", op1, op1 & mask);

    return 0;
}
```

The result is

```
$ g++ main.cpp -o test1
$ ./test1
a (AND) mask = A
a   &   0xDF = A
```

Chapter 7: Bit fields

Bit fields tightly pack C and C++ structures to reduce size. This appears painless: specify the number of bits for members, and compiler does the work of co-mingling bits. The restriction is inability to take the address of a bit field member, since it is stored co-mingled. `sizeof()` is also disallowed.

The cost of bit fields is slower access, as memory must be retrieved and bitwise operations applied to extract or modify member values. These operations also add to executable size.

Section 7.1: Declaration and Usage

```
struct FileAttributes
{
    unsigned int ReadOnly: 1;
    unsigned int Hidden: 1;
};
```

Here, each of these two fields will occupy 1 bit in memory. It is specified by `: 1` expression after the variable names. Base type of bit field could be any integral type (8-bit int to 64-bit int). Using `unsigned` type is recommended, otherwise surprises may come.

If more bits are required, replace "1" with number of bits required. For example:

```
struct Date
{
    unsigned int Year : 13; // 2^13 = 8192, enough for "year" representation for long time
    unsigned int Month: 4;  // 2^4 = 16, enough to represent 1-12 month values.
    unsigned int Day:   5;  // 32
};
```

The whole structure is using just 22 bits, and with normal compiler settings, `sizeof` this structure would be 4 bytes.

Usage is pretty simple. Just declare the variable, and use it like ordinary structure.

```
Date d;

d.Year = 2016;
d.Month = 7;
d.Day =  22;

std::cout << "Year:" << d.Year << std::endl <<
        "Month:" << d.Month << std::endl <<
        "Day:" << d.Day << std::endl;
```

Chapter 8: Arrays

Arrays are elements of the same type placed in adjoining memory locations. The elements can be individually referenced by a unique identifier with an added index.

This allows you to declare multiple variable values of a specific type and access them individually without needing to declare a variable for each value.

Section 8.1: Array initialization

An array is just a block of sequential memory locations for a specific type of variable. Arrays are allocated the same way as normal variables, but with square brackets appended to its name `[]` that contain the number of elements that fit into the array memory.

The following example of an array uses the typ `int`, the variable name `arrayOfInts`, and the number of elements `[5]` that the array has space for:

```
int arrayOfInts[5];
```

An array can be declared and initialized at the same time like this

```
int arrayOfInts[5] = {10, 20, 30, 40, 50};
```

When initializing an array by listing all of its members, it is not necessary to include the number of elements inside the square brackets. It will be automatically calculated by the compiler. In the following example, it's 5:

```
int arrayOfInts[] = {10, 20, 30, 40, 50};
```

It is also possible to initialize only the first elements while allocating more space. In this case, defining the length in brackets is mandatory. The following will allocate an array of length 5 with partial initialization, the compiler initializes all remaining elements with the standard value of the element type, in this case zero.

```
int arrayOfInts[5] = {10,20}; // means 10, 20, 0, 0, 0
```

Arrays of other basic data types may be initialized in the same way.

```
char arrayOfChars[5]; // declare the array and allocate the memory, don't initialize

char arrayOfChars[5] = { 'a', 'b', 'c', 'd', 'e' } ; //declare and initialize

double arrayOfDoubles[5] = {1.14159, 2.14159, 3.14159, 4.14159, 5.14159};

string arrayOfStrings[5] = { "C++", "is", "super", "duper", "great!"};
```

It is also important to take note that when accessing array elements, the array's element index(or position) starts from 0.

```
int array[5] = { 10/*Element no.0*/, 20/*Element no.1*/, 30, 40, 50/*Element no.4*/};
std::cout << array[4]; //outputs 50
std::cout << array[0]; //outputs 10
```

Section 8.2: A fixed size raw array matrix (that is, a 2D raw array)

```
// A fixed size raw array matrix (that is, a 2D raw array).
#include <iostream>
#include <iomanip>
using namespace std;

auto main() -> int
{
    int const   n_rows  = 3;
    int const   n_cols  = 7;
    int const   m[n_rows][n_cols] =             // A raw array matrix.
    {
        {  1,  2,  3,  4,  5,  6,  7 },
        {  8,  9, 10, 11, 12, 13, 14 },
        { 15, 16, 17, 18, 19, 20, 21 }
    };

    for( int y = 0; y < n_rows; ++y )
    {
        for( int x = 0; x < n_cols; ++x )
        {
            cout << setw( 4 ) << m[y][x];       // Note: do NOT use m[y,x]!
        }
        cout << '\n';
    }
}
```

Output:

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21

C++ doesn't support special syntax for indexing a multi-dimensional array. Instead such an array is viewed as an array of arrays (possibly of arrays, and so on), and the ordinary single index notation `[i]` is used for each level. In the example above `m[y]` refers to row `y` of `m`, where `y` is a zero-based index. Then this row can be indexed in turn, e.g. `m[y][x]`, which refers to the xth item – or column – of row y.

I.e. the last index varies fastest, and in the declaration the range of this index, which here is the number of columns per row, is the last and “innermost” size specified.

Since C++ doesn't provide built-in support for dynamic size arrays, other than dynamic allocation, a dynamic size matrix is often implemented as a class. Then the raw array matrix indexing notation `m[y][x]` has some cost, either by exposing the implementation (so that e.g. a view of a transposed matrix becomes practically impossible) or by adding some overhead and slight inconvenience when it's done by returning a proxy object from `operator[]`. And so the indexing notation for such an abstraction can and will usually be different, both in look-and-feel and in the order of indices, e.g. `m(x,y)` or `m.at(x,y)` or `m.item(x,y)`.

Section 8.3: Dynamically sized raw array

```
// Example of raw dynamic size array. It's generally better to use std::vector.
#include <algorithm>            // std::sort
#include <iostream>
using namespace std;

auto int_from( istream& in ) -> int { int x; in >> x; return x; }
```

```
auto main()
    -> int
{
    cout << "Sorting n integers provided by you.\\n";
    cout << "n? ";
    int const   n   = int_from( cin );
    int*        a   = new int[n];       // ← Allocation of array of n items.

    for( int i = 1; i <= n; ++i )
    {
        cout << "The #" << i << " number, please: ";
        a[i-1] = int_from( cin );
    }

    sort( a, a + n );
    for( int i = 0; i < n; ++i ) { cout << a[i] << ' '; }
    cout << '\\n';

    delete[] a;
}
```

A program that declares an array `T a[n];` where `n` is determined a run-time, can compile with certain compilers that support C99 *variadic length arrays* (VLAs) as a language extension. But VLAs are not supported by standard C++. This example shows how to manually allocate a dynamic size array via a `new[]`-expression,

```
int*        a   = new int[n];       // ← Allocation of array of n items.
```

… then use it, and finally deallocate it via a `delete[]`-expression:

```
delete[] a;
```

The array allocated here has indeterminate values, but it can be zero-initialized by just adding an empty parenthesis `()`, like this: `new int[n]()`. More generally, for arbitrary item type, this performs a *value-initialization*.

As part of a function down in a call hierarchy this code would not be exception safe, since an exception before the `delete[]` expression (and after the `new[]`) would cause a memory leak. One way to address that issue is to automate the cleanup via e.g. a `std::unique_ptr` smart pointer. But a generally better way to address it is to just use a `std::vector`: that's what `std::vector` is there for.

Section 8.4: Array size: type safe at compile time

```
#include      // size_t, ptrdiff_t

//----------------------------------- Machinery:

using Size = ptrdiff_t;

template< class Item, size_t n >
constexpr auto n_items( Item (&)[n] ) noexcept
-> Size
{ return n; }


//----------------------------------- Usage:

#include
using namespace std;
auto main()
```

```
-> int
{
int const   a[]     = {3, 1, 4, 1, 5, 9, 2, 6, 5, 4};
Size const  n       = n_items( a );
int         b[n]    = {};       // An array of the same size as a.

(void) b;
cout <}
```

The C idiom for array size, `sizeof(a)/sizeof(a[0])`, will accept a pointer as argument and will then generally yield an incorrect result.

For C++11

using C++11 you can do:

```
std::extent<decltype(MyArray)>::value;
```

Example:

```
char MyArray[] = { 'X','o','c','e' };
const auto n = std::extent<decltype(MyArray)>::value;
std::cout << n << "\n"; // Prints 4
```

Up till C++17 (forthcoming as of this writing) C++ had no built-in core language or standard library utility to obtain the size of an array, but this can be implemented by passing the array *by reference* to a function template, as shown above. Fine but important point: the template size parameter is a `size_t`, somewhat inconsistent with the signed `Size` function result type, in order to accommodate the g++ compiler which sometimes insists on `size_t` for template matching.

With C++17 and later one may instead use `std::size`, which is specialized for arrays.

Section 8.5: Expanding dynamic size array by using std::vector

```
// Example of std::vector as an expanding dynamic size array.
#include <algorithm>            // std::sort
#include <iostream>
#include <vector>               // std::vector
using namespace std;

int int_from( std::istream& in ) { int x = 0; in >> x; return x; }

int main()
{
    cout << "Sorting integers provided by you.\n";
    cout << "You can indicate EOF via F6 in Windows or Ctrl+D in Unix-land.\n";
    vector<int> a;      // ← Zero size by default.

    while( cin )
    {
        cout << "One number, please, or indicate EOF: ";
        int const x = int_from( cin );
        if( !cin.fail() ) { a.push_back( x ); }  // Expands as necessary.
    }

    sort( a.begin(), a.end() );
```

```
    int const n = a.size();
    for( int i = 0; i < n; ++i ) { cout << a[i] << ' '; }
    cout << '\n';
}
```

`std::vector` is a standard library class template that provides the notion of a variable size array. It takes care of all the memory management, and the buffer is contiguous so a pointer to the buffer (e.g. `&v[0]` or `v.data()`) can be passed to API functions requiring a raw array. A `vector` can even be expanded at run time, via e.g. the `push_back` member function that appends an item.

The complexity of the sequence of *n* `push_back` operations, including the copying or moving involved in the vector expansions, is amortized O(*n*). "Amortized": on average.

Internally this is usually achieved by the vector *doubling* its buffer size, its capacity, when a larger buffer is needed. E.g. for a buffer starting out as size 1, and being repeatedly doubled as needed for *n*=17 `push_back` calls, this involves 1 + 2 + 4 + 8 + 16 = 31 copy operations, which is less than 2×*n* = 34. And more generally the sum of this sequence can't exceed 2×*n*.

Compared to the dynamic size raw array example, this `vector`-based code does not require the user to supply (and know) the number of items up front. Instead the vector is just expanded as necessary, for each new item value specified by the user.

Section 8.6: A dynamic size matrix using std::vector for storage

Unfortunately as of C++14 there's no dynamic size matrix class in the C++ standard library. Matrix classes that support dynamic size are however available from a number of 3rd party libraries, including the Boost Matrix library (a sub-library within the Boost library).

If you don't want a dependency on Boost or some other library, then one poor man's dynamic size matrix in C++ is just like

```
vector<vector<int>> m( 3, vector<int>( 7 ) );
```

… where `vector` is `std::vector`. The matrix is here created by copying a row vector *n* times where *n* is the number of rows, here 3. It has the advantage of providing the same `m[y][x]` indexing notation as for a fixed size raw array matrix, but it's a bit inefficient because it involves a dynamic allocation for each row, and it's a bit unsafe because it's possible to inadvertently resize a row.

A more safe and efficient approach is to use a single vector as *storage* for the matrix, and map the client code's (*x*, *y*) to a corresponding index in that vector:

```
// A dynamic size matrix using std::vector for storage.

//--------------------------------------------- Machinery:
#include         // std::copy
#include          // assert
#include  // std::initializer_list
#include            // std::vector
#include          // ptrdiff_t

namespace my {
using Size = ptrdiff_t;
using std::initializer_list;
using std::vector;
```

```
template< class Item >
class Matrix
{
private:
vector    items_;
Size            n_cols_;

auto index_for( Size const x, Size const y ) const
-> Size
{ return y*n_cols_ + x; }

public:
auto n_rows() const -> Size { return items_.size()/n_cols_; }
auto n_cols() const -> Size { return n_cols_; }

auto item( Size const x, Size const y )
-> Item&
{ return items_[index_for(x, y)]; }

auto item( Size const x, Size const y ) const
-> Item const&
{ return items_[index_for(x, y)]; }

Matrix(): n_cols_( 0 ) {}

Matrix( Size const n_cols, Size const n_rows )
: items_( n_cols*n_rows )
, n_cols_( n_cols )
{}

Matrix( initializer_list< initializer_list > const& values )
: items_()
, n_cols_( values.size() == 0? 0 : values.begin()->size() )
{
for( auto const& row : values )
{
assert( Size( row.size() ) == n_cols_ );
items_.insert( items_.end(), row.begin(), row.end() );
}
}
};
}  // namespace my

//-------------------------------------------- Usage:
using my::Matrix;

auto some_matrix()
-> Matrix
{
return
{
{  1,  2,  3,  4,  5,  6,  7 },
{  8,  9, 10, 11, 12, 13, 14 },
{ 15, 16, 17, 18, 19, 20, 21 }
};
}

#include
#include
using namespace std;
auto main() -> int
{
Matrix const m = some_matrix();
assert( m.n_cols() == 7 );
```

```
assert( m.n_rows() == 3 );
for( int y = 0, y_end = m.n_rows(); y < y_end; ++y )
{
for( int x = 0, x_end = m.n_cols(); x < x_end; ++x )
{
cout <← Note: not `m[y][x]`!
}
cout <}
}
```

Output:

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21

The above code is not industrial grade: it's designed to show the basic principles, and serve the needs of students learning C++.

For example, one may define `operator()` overloads to simplify the indexing notation.

Chapter 9: Iterators

Section 9.1: Overview

Iterators are Positions

Iterators are a means of navigating and operating on a sequence of elements and are a generalized extension of pointers. Conceptually it is important to remember that iterators are positions, not elements. For example, take the following sequence:

```
A B C
```

The sequence contains three elements and four positions

```
+---+---+---+---+
| A | B | C |   |
+---+---+---+---+
```

Elements are things within a sequence. Positions are places where meaningful operations can happen to the sequence. For example, one inserts into a position, *before* or *after* element A, not into an element. Even deletion of an element (`erase(A)`) is done by first finding its position, then deleting it.

From Iterators to Values

To convert from a position to a value, an iterator is *dereferenced*:

```
auto my_iterator = my_vector.begin(); // position
auto my_value = *my_iterator; // value
```

One can think of an iterator as dereferencing to the value it refers to in the sequence. This is especially useful in understanding why you should never dereference the `end()` iterator in a sequence:

```
+---+---+---+---+
| A | B | C |   |
+---+---+---+---+
↑           ↑
|           +-- An iterator here has no value. Do not dereference it!
+-------------- An iterator here dereferences to the value A.
```

In all the sequences and containers found in the C++ standard library, `begin()` will return an iterator to the first position, and `end()` will return an iterator to one past the last position (*not* the last position!). Consequently, the names of these iterators in algorithms are oftentimes labelled `first` and `last`:

```
+---+---+---+---+
| A | B | C |   |
+---+---+---+---+
↑           ↑
|           |
+- first    +- last
```

It is also possible to obtain an iterator to *any sequence*, because even an empty sequence contains at least one

position:

```
+---+
|   |
+---+
```

In an empty sequence, `begin()` and `end()` will be the same position, and *neither* can be dereferenced:

```
+---+
|   |
+---+
  ↑
  |
  +- empty_sequence.begin()
  |
  +- empty_sequence.end()
```

The alternative visualization of iterators is that they mark the positions *between* elements:

```
+---+---+---+
| A | B | C |
+---+---+---+
↑   ^   ^   ↑
|           |
+- first    +- last
```

and dereferencing an iterator returns a reference to the element coming after the iterator. Some situations where this view is particularly useful are:

- `insert` operations will insert elements into the position indicated by the iterator,
- `erase` operations will return an iterator corresponding to the same position as the one passed in,
- an iterator and its corresponding reverse iterator are located in the same .position between elements

Invalid Iterators

An iterator becomes *invalidated* if (say, in the course of an operation) its position is no longer a part of a sequence. An invalidated iterator cannot be dereferenced until it has been reassigned to a valid position. For example:

```
std::vector<int>::iterator first;
{
    std::vector<int> foo;
    first = foo.begin(); // first is now valid
} // foo falls out of scope and is destroyed
// At this point first is now invalid
```

The many algorithms and sequence member functions in the C++ standard library have rules governing when iterators are invalidated. Each algorithm is different in the way they treat (and invalidate) iterators.

Navigating with Iterators

As we know, iterators are for navigating sequences. In order to do that an iterator must migrate its position throughout the sequence. Iterators can advance forward in the sequence and some can advance backwards:

```
auto first = my_vector.begin();
++first;                                             // advance the iterator 1 position
```

```
std::advance(first, 1);                          // advance the iterator 1 position
first = std::next(first);                        // returns iterator to the next element
std::advance(first, -1);                         // advance the iterator 1 position backwards
first = std::next(first, 20);                    // returns iterator to the element 20 position
forward
first = std::prev(first, 5);                     // returns iterator to the element 5 position
backward
auto dist = std::distance(my_vector.begin(), first); // returns distance between two iterators.
```

Note, second argument of std::distance should be reachable from the first one(or, in other words `first` should be less or equal than `second`).

Even though you can perform arithmetic operators with iterators, not all operations are defined for all types of iterators. `a = b + 3;` would work for Random Access Iterators, but wouldn't work for Forward or Bidirectional Iterators, which still can be advanced by 3 position with something like `b = a; ++b; ++b; ++b;`. So it is recommended to use special functions in case you are not sure what is iterator type (for example, in a template function accepting iterator).

Iterator Concepts

The C++ standard describes several different iterator concepts. These are grouped according to how they behave in the sequences they refer to. If you know the concept an iterator *models* (behaves like), you can be assured of the behavior of that iterator *regardless of the sequence to which it belongs*. They are often described in order from the most to least restrictive (because the next iterator concept is a step better than its predecessor):

- Input Iterators : Can be dereferenced *only once* per position. Can only advance, and only one position at a time.
- Forward Iterators : An input iterator that can be dereferenced any number of times.
- Bidirectional Iterators : A forward iterator that can also advance *backwards* one position at a time.
- Random Access Iterators : A bidirectional iterator that can advance forwards or backwards any number of positions at a time.
- Contiguous Iterators (since C++17) : A random access iterator that guaranties that underlying data is contiguous in memory.

Algorithms can vary depending on the concept modeled by the iterators they are given. For example, although `random_shuffle` can be implemented for forward iterators, a more efficient variant that requires random access iterators could be provided.

Iterator traits

Iterator traits provide uniform interface to the properties of iterators. They allow you to retrieve value, difference, pointer, reference types and also category of iterator:

```
template<class Iter>
Iter find(Iter first, Iter last, typename std::iterator_traits<Iter>::value_type val)  {
    while (first != last) {
        if (*first == val)
            return first;
        ++first;
    }
    return last;
}
```

Category of iterator can be used to specialize algorithms:

```
template<class BidirIt>
```

```
void test(BidirIt a, std::bidirectional_iterator_tag)  {
    std::cout << "Bidirectional iterator is used" << std::endl;
}

template<class ForwIt>
void test(ForwIt a, std::forward_iterator_tag)  {
    std::cout << "Forward iterator is used" << std::endl;
}

template<class Iter>
void test(Iter a)  {
    test(a, typename std::iterator_traits<Iter>::iterator_category());
}
```

Categories of iterators are basically iterators concepts, except Contiguous Iterators don't have their own tag, since it was found to break code.

Section 9.2: Vector Iterator

`begin` returns an `iterator` to the first element in the sequence container.

`end` returns an `iterator` to the first element past the end.

If the vector object is `const`, both `begin` and `end` return a `const_iterator`. If you want a `const_iterator` to be returned even if your vector is not `const`, you can use `cbegin` and `cend`.

Example:

```
#include <vector>
#include <iostream>

int main() {
    std::vector<int> v = { 1, 2, 3, 4, 5 };  //intialize vector using an initializer_list

    for (std::vector<int>::iterator it = v.begin(); it != v.end(); ++it) {
        std::cout << *it << " ";
    }

    return 0;
}
```

Output:

> 1 2 3 4 5

Section 9.3: Map Iterator

An iterator to the first element in the container.

If a map object is const-qualified, the function returns a `const_iterator`. Otherwise, it returns an `iterator`.

```
// Create a map and insert some values
std::map<char,int> mymap;
mymap['b'] = 100;
mymap['a'] = 200;
mymap['c'] = 300;
```

```
// Iterate over all tuples
for (std::map<char,int>::iterator it = mymap.begin(); it != mymap.end(); ++it)
    std::cout << it->first << " => " << it->second << '\n';
```

Output:

> a => 200
> b => 100
> c => 300

Section 9.4: Reverse Iterators

If we want to iterate backwards through a list or vector we can use a `reverse_iterator`. A reverse iterator is made from a bidirectional, or random access iterator which it keeps as a member which can be accessed through `base()`.

To iterate backwards use `rbegin()` and `rend()` as the iterators for the end of the collection, and the start of the collection respectively.

For instance, to iterate backwards use:

```
std::vector<int> v{1, 2, 3, 4, 5};
for (std::vector<int>::reverse_iterator it = v.rbegin(); it != v.rend(); ++it)
{
    cout << *it;
} // prints 54321
```

A reverse iterator can be converted to a forward iterator via the `base()` member function. The relationship is that the reverse iterator references one element past the `base()` iterator:

```
std::vector<int>::reverse_iterator r = v.rbegin();
std::vector<int>::iterator i = r.base();
assert(&*r == &*(i-1)); // always true if r, (i-1) are dereferenceable
                        // and are not proxy iterators

 +---+---+---+---+---+---+---+
 |   | 1 | 2 | 3 | 4 | 5 |   |
 +---+---+---+---+---+---+---+
   ↑   ↑               ↑   ↑
   |   |               |   |
rend() |         rbegin()  end()
       |                   rbegin().base()
     begin()
     rend().base()
```

In the visualization where iterators mark positions between elements, the relationship is simpler:

```
  +---+---+---+---+---+
| 1 | 2 | 3 | 4 | 5 |
+---+---+---+---+---+
↑                   ↑
|                   |
|                 end()
|                 rbegin()
begin()             rbegin().base()
rend()
rend().base()
```

Section 9.5: Stream Iterators

Stream iterators are useful when we need to read a sequence or print formatted data from a container:

```
// Data stream. Any number of various whitespace characters will be OK.
std::istringstream istr("1\t 2     3 4");
std::vector<int> v;

// Constructing stream iterators and copying data from stream into vector.
std::copy(
    // Iterator which will read stream data as integers.
    std::istream_iterator<int>(istr),
    // Default constructor produces end-of-stream iterator.
    std::istream_iterator<int>(),
    std::back_inserter(v));

// Print vector contents.
std::copy(v.begin(), v.end(),
    //Will print values to standard output as integers delimeted by " -- ".
    std::ostream_iterator<int>(std::cout, " -- "));
```

The example program will print `1 -- 2 -- 3 -- 4 --` to standard output.

Section 9.6: C Iterators (Pointers)

```
// This creates an array with 5 values.
const int array[] = { 1, 2, 3, 4, 5 };

#ifdef BEFORE_CPP11

// You can use `sizeof` to determine how many elements are in an array.
const int* first = array;
const int* afterLast = first + sizeof(array) / sizeof(array[0]);

// Then you can iterate over the array by incrementing a pointer until
// it reaches past the end of our array.
for (const int* i = first; i < afterLast; ++i) {
    std::cout << *i << std::endl;
}

#else

// With C++11, you can let the STL compute the start and end iterators:
for (auto i = std::begin(array); i != std::end(array); ++i) {
    std::cout << *i << std::endl;
}

#endif
```

This code would output the numbers 1 through 5, one on each line like this:

> 1
> 2
> 3
> 4

```
5
```

Breaking It Down

```
const int array[] = { 1, 2, 3, 4, 5 };
```

This line creates a new integer array with 5 values. C arrays are just pointers to memory where each value is stored together in a contiguous block.

```
const int* first = array;
const int* afterLast = first + sizeof(array) / sizeof(array[0]);
```

These lines create two pointers. The first pointer is given the value of the array pointer, which is the address of the first element in the array. The `sizeof` operator when used on a C array returns the size of the array in bytes. Divided by the size of an element this gives the number of elements in the array. We can use this to find the address of the block *after* the array.

```
for (const int* i = first; i < afterLast; ++i) {
```

Here we create a pointer which we will use as an iterator. It is initialized with the address of the first element we want to iterate over, and we'll continue to iterate as long as `i` is less than `afterLast`, which means as long as `i` is pointing to an address within `array`.

```
    std::cout << *i << std::endl;
```

Finally, within the loop we can access the value our iterator `i` is pointing to by dereferencing it. Here the dereference operator `*` returns the value at the address in `i`.

Section 9.7: Write your own generator-backed iterator

A common pattern in other languages is having a function that produces a "stream" of objects, and being able to use loop-code to loop over it.

We can model this in C++ as

```
template<class T>
struct generator_iterator {
  using difference_type=std::ptrdiff_t;
  using value_type=T;
  using pointer=T*;
  using reference=T;
  using iterator_category=std::input_iterator_tag;
  std::optional<T> state;
  std::function< std::optional<T>() > operation;
  // we store the current element in "state" if we have one:
  T operator*() const {
    return *state;
  }
  // to advance, we invoke our operation.  If it returns a nullopt
  // we have reached the end:
  generator_iterator& operator++() {
    state = operation();
    return *this;
  }
  generator_iterator operator++(int) {
    auto r = *this;
```

```
    ++(*this);
    return r;
  }
  // generator iterators are only equal if they are both in the "end" state:
  friend bool operator==( generator_iterator const& lhs, generator_iterator const& rhs ) {
    if (!lhs.state && !rhs.state) return true;
    return false;
  }
  friend bool operator!=( generator_iterator const& lhs, generator_iterator const& rhs ) {
    return !(lhs==rhs);
  }
  // We implicitly construct from a std::function with the right signature:
  generator_iterator( std::function< std::optional<T>() > f ):operation(std::move(f))
  {
    if (operation)
      state = operation();
  }
  // default all special member functions:
  generator_iterator( generator_iterator && ) =default;
  generator_iterator( generator_iterator const& ) =default;
  generator_iterator& operator=( generator_iterator && ) =default;
  generator_iterator& operator=( generator_iterator const& ) =default;
  generator_iterator() =default;
};
```

live example.

We store the generated element early so we can more easily detect if we are already at the end.

As the function of an end generator iterator is never used, we can create a range of generator iterators by only copying the `std::function` once. A default constructed generator iterator compares equal to itself, and to all other end-generator-iterators.

Chapter 10: Basic input/output in c++

Section 10.1: user input and standard output

```
#include <iostream>

int main()
{
    int value;
    std::cout << "Enter a value: " << std::endl;
    std::cin >> value;
    std::cout << "The square of entered value is: " << value * value << std::endl;
    return 0;
}
```

Chapter 11: Loops

A loop statement executes a group of statements repeatedly until a condition is met. There are 3 types of primitive loops in C++: for, while, and do...while.

Section 11.1: Range-Based For

Version ≥ C++11

`for` loops can be used to iterate over the elements of a iterator-based range, without using a numeric index or directly accessing the iterators:

```
vector<float> v = {0.4f, 12.5f, 16.234f};

for(auto val: v)
{
    std::cout << val << " ";
}

std::cout << std::endl;
```

This will iterate over every element in `v`, with `val` getting the value of the current element. The following statement:

```
for (for-range-declaration : for-range-initializer ) statement
```

is equivalent to:

```
{
    auto&& __range = for-range-initializer;
    auto __begin = begin-expr, __end = end-expr;
    for (; __begin != __end; ++__begin) {
        for-range-declaration = *__begin;
        statement
    }
}
```

Version ≥ C++17

```
{
    auto&& __range = for-range-initializer;
    auto __begin = begin-expr;
    auto __end = end-expr; // end is allowed to be a different type than begin in C++17
    for (; __begin != __end; ++__begin) {
        for-range-declaration = *__begin;
        statement
    }
}
```

This change was introduced for the planned support of Ranges TS in C++20.

In this case, our loop is equivalent to:

```
{
    auto&& __range = v;
    auto __begin = v.begin(), __end = v.end();
    for (; __begin != __end; ++__begin) {
        auto val = *__begin;
        std::cout << val << " ";
```

```
    }
}
```

Note that `auto val` declares a value type, which will be a copy of a value stored in the range (we are copy-initializing it from the iterator as we go). If the values stored in the range are expensive to copy, you may want to use `const auto &val`. You are also not required to use `auto`; you can use an appropriate typename, so long as it is implicitly convertible from the range's value type.

If you need access to the iterator, range-based for cannot help you (not without some effort, at least).

If you wish to reference it, you may do so:

```
vector<float> v = {0.4f, 12.5f, 16.234f};

for(float &val: v)
{
    std::cout << val << " ";
}
```

You could iterate on `const` reference if you have `const` container:

```
const vector<float> v = {0.4f, 12.5f, 16.234f};

for(const float &val: v)
{
    std::cout << val << " ";
}
```

One would use forwarding references when the sequence iterator returns a proxy object and you need to operate on that object in a non-`const` way. Note: it will most likely confuse readers of your code.

```
vector<bool> v(10);

for(auto&& val: v)
{
    val = true;
}
```

The "range" type provided to range-based `for` can be one of the following:

- Language arrays:

  ```
  float arr[] = {0.4f, 12.5f, 16.234f};

  for(auto val: arr)
  {
      std::cout << val << " ";
  }
  ```

  Note that allocating a dynamic array does not count:

  ```
  float *arr = new float[3]{0.4f, 12.5f, 16.234f};

  for(auto val: arr) //Compile error.
  {
      std::cout << val << " ";
  }
  ```

- Any type which has member functions `begin()` and `end()`, which return iterators to the elements of the type. The standard library containers qualify, but user-defined types can be used as well:

```
struct Rng
{
    float arr[3];

    // pointers are iterators
    const float* begin() const {return &arr[0];}
    const float* end() const   {return &arr[3];}
    float* begin() {return &arr[0];}
    float* end()   {return &arr[3];}
};

int main()
{
    Rng rng = {{0.4f, 12.5f, 16.234f}};

    for(auto val: rng)
    {
        std::cout << val << " ";
    }
}
```

- Any type which has non-member `begin(type)` and `end(type)` functions which can found via argument dependent lookup, based on `type`. This is useful for creating a range type without having to modify class type itself:

```
namespace Mine
{
    struct Rng {float arr[3];};

    // pointers are iterators
    const float* begin(const Rng &rng) {return &rng.arr[0];}
    const float* end(const Rng &rng) {return &rng.arr[3];}
    float* begin(Rng &rng) {return &rng.arr[0];}
    float* end(Rng &rng) {return &rng.arr[3];}
}

int main()
{
    Mine::Rng rng = {{0.4f, 12.5f, 16.234f}};

    for(auto val: rng)
    {
        std::cout << val << " ";
    }
}
```

Section 11.2: For loop

A `for` loop executes statements in the `loop body`, while the loop `condition` is true. Before the loop `initialization statement` is executed exactly once. After each cycle, the `iteration execution` part is executed.

A `for` loop is defined as follows:

```
for (/*initialization statement*/; /*condition*/; /*iteration execution*/)
```

```
{
    // body of the loop
}
```

Explanation of the placeholder statements:

- `initialization statement`: This statement gets executed only once, at the beginning of the `for` loop. You can enter a declaration of multiple variables of one type, such as `int i = 0, a = 2, b = 3`. These variables are only valid in the scope of the loop. Variables defined before the loop with the same name are hidden during execution of the loop.
- `condition`: This statement gets evaluated ahead of each *loop body* execution, and aborts the loop if it evaluates to `false`.
- `iteration execution`: This statement gets executed after the loop *body*, ahead of the next *condition* evaluation, unless the `for` loop is aborted in the *body* (by `break`, `goto`, `return` or an exception being thrown). You can enter multiple statements in the `iteration execution` part, such as `a++, b+=10, c=b+a`.

The rough equivalent of a `for` loop, rewritten as a `while` loop is:

```
/*initialization*/
while (/*condition*/)
{
    // body of the loop; using 'continue' will skip to increment part below
    /*iteration execution*/
}
```

The most common case for using a `for` loop is to execute statements a specific number of times. For example, consider the following:

```
for(int i = 0; i < 10; i++) {
    std::cout << i << std::endl;
}
```

A valid loop is also:

```
for(int a = 0, b = 10, c = 20; (a+b+c < 100); c--, b++, a+=c) {
    std::cout << a << " " << b << " " << c << std::endl;
}
```

An example of hiding declared variables before a loop is:

```
int i = 99; //i = 99
for(int i = 0; i < 10; i++) { //we declare a new variable i
    //some operations, the value of i ranges from 0 to 9 during loop execution
}
//after the loop is executed, we can access i with value of 99
```

But if you want to use the already declared variable and not hide it, then omit the declaration part:

```
int i = 99; //i = 99
for(i = 0; i < 10; i++) { //we are using already declared variable i
    //some operations, the value of i ranges from 0 to 9 during loop execution
}
//after the loop is executed, we can access i with value of 10
```

Notes:

- The initialization and increment statements can perform operations unrelated to the condition statement, or nothing at all - if you wish to do so. But for readability reasons, it is best practice to only perform operations directly relevant to the loop.
- A variable declared in the initialization statement is visible only inside the scope of the `for` loop and is released upon termination of the loop.
- Don't forget that the variable which was declared in the `initialization statement` can be modified during the loop, as well as the variable checked in the `condition`.

Example of a loop which counts from 0 to 10:

```
for (int counter = 0; counter <= 10; ++counter)
{
    std::cout << counter << '\n';
}
// counter is not accessible here (had value 11 at the end)
```

Explanation of the code fragments:

- `int counter = 0` initializes the variable `counter` to 0. (This variable can only be used inside of the `for` loop.)
- `counter <= 10` is a Boolean condition that checks whether `counter` is less than or equal to 10. If it is `true`, the loop executes. If it is `false`, the loop ends.
- `++counter` is an increment operation that increments the value of `counter` by 1 ahead of the next condition check.

By leaving all statements empty, you can create an infinite loop:

```
// infinite loop
for (;;)
    std::cout << "Never ending!\n";
```

The `while` loop equivalent of the above is:

```
// infinite loop
while (true)
    std::cout << "Never ending!\n";
```

However, an infinite loop can still be left by using the statements `break`, `goto`, or `return` or by throwing an exception.

The next common example of iterating over all elements from an STL collection (e.g., a `vector`) without using the **`<algorithm>`** header is:

```
std::vector<std::string> names = {"Albert Einstein", "Stephen Hawking", "Michael Ellis"};
for(std::vector<std::string>::iterator it = names.begin(); it != names.end(); ++it) {
    std::cout << *it << std::endl;
}
```

Section 11.3: While loop

A `while` loop executes statements repeatedly until the given condition evaluates to `false`. This control statement is used when it is not known, in advance, how many times a block of code is to be executed.

For example, to print all the numbers from 0 up to 9, the following code can be used:

```
int i = 0;
```

```
while (i < 10)
{
    std::cout << i << " ";
    ++i; // Increment counter
}
std::cout << std::endl; // End of line; "0 1 2 3 4 5 6 7 8 9" is printed to the console
```

Version ≥ C++17

Note that since C++17, the first 2 statements can be combined

```
while (int i = 0; i < 10)
//... The rest is the same
```

To create an infinite loop, the following construct can be used:

```
while (true)
{
    // Do something forever (however, you can exit the loop by calling 'break'
}
```

There is another variant of `while` loops, namely the `do...while` construct. See the do-while loop example for more information.

Section 11.4: Do-while loop

A *do-while* loop is very similar to a *while* loop, except that the condition is checked at the end of each cycle, not at the start. The loop is therefore guaranteed to execute at least once.

The following code will print `0`, as the condition will evaluate to `false` at the end of the first iteration:

```
int i =0;
do
{
    std::cout << i;
    ++i; // Increment counter
}
while (i < 0);
std::cout << std::endl; // End of line; 0 is printed to the console
```

Note: Do not forget the semicolon at the end of `while(condition);`, which is needed in the *do-while* construct.

In contrast to the *do-while* loop, the following will not print anything, because the condition evaluates to `false` at the beginning of the first iteration:

```
int i =0;
while (i < 0)
{
    std::cout << i;
    ++i; // Increment counter
}
std::cout << std::endl; // End of line; nothing is printed to the console
```

Note: A *while* loop can be exited without the condition becoming false by using a `break`, `goto`, or `return` statement.

```
int i = 0;
do
{
```

```
    std::cout << i;
    ++i; // Increment counter
    if (i > 5)
    {
        break;
    }
}
while (true);
std::cout << std::endl; // End of line; 0 1 2 3 4 5 is printed to the console
```

A trivial *do-while* loop is also occasionally used to write macros that require their own scope (in which case the trailing semicolon is omitted from the macro definition and required to be provided by the user):

```
#define BAD_MACRO(x) f1(x); f2(x); f3(x);

// Only the call to f1 is protected by the condition here
if (cond) BAD_MACRO(var);

#define GOOD_MACRO(x) do { f1(x); f2(x); f3(x); } while(0)

// All calls are protected here
if (cond) GOOD_MACRO(var);
```

Section 11.5: Loop Control statements : Break and Continue

Loop control statements are used to change the flow of execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed. The `break` and `continue` are loop control statements.

The `break` statement terminates a loop without any further consideration.

```
for (int i = 0; i < 10; i++)
{
    if (i == 4)
        break; // this will immediately exit our loop
    std::cout << i << '\n';
}
```

The above code will print out:

```
1
2
3
```

The `continue` statement does not immediately exit the loop, but rather skips the rest of the loop body and goes to the top of the loop (including checking the condition).

```
for (int i = 0; i < 6; i++)
{
    if (i % 2 == 0) // evaluates to true if i is even
        continue; // this will immediately go back to the start of the loop
    /* the next line will only be reached if the above "continue" statement
       does not execute  */
    std::cout << i << " is an odd number\n";
}
```

The above code will print out:

```
1 is an odd number
3 is an odd number
5 is an odd number
```

Because such control flow changes are sometimes difficult for humans to easily understand, `break` and `continue` are used sparingly. More straightforward implementation are usually easier to read and understand. For example, the first `for` loop with the `break` above might be rewritten as:

```
for (int i = 0; i < 4; i++)
{
    std::cout << i << '\n';
}
```

The second example with `continue` might be rewritten as:

```
for (int i = 0; i < 6; i++)
{
    if (i % 2 != 0) {
        std::cout << i << " is an odd number\n";
    }
}
```

Section 11.6: Declaration of variables in conditions

In the condition of the `for` and `while` loops, it's also permitted to declare an object. This object will be considered to be in scope until the end of the loop, and will persist through each iteration of the loop:

```
for (int i = 0; i < 5; ++i) {
    do_something(i);
}
// i is no longer in scope.

for (auto& a : some_container) {
    a.do_something();
}
// a is no longer in scope.

while(std::shared_ptr<Object> p = get_object()) {
   p->do_something();
}
// p is no longer in scope.
```

However, it is not permitted to do the same with a `do...while` loop; instead, declare the variable before the loop, and (optionally) enclose both the variable and the loop within a local scope if you want the variable to go out of scope after the loop ends:

```
//This doesn't compile
do {
    s = do_something();
} while (short s > 0);

// Good
short s;
do {
    s = do_something();
} while (s > 0);
```

This is because the *statement* portion of a `do...while` loop (the loop's body) is evaluated before the *expression* portion (the `while`) is reached, and thus, any declaration in the *expression* will not be visible during the first iteration of the loop.

Section 11.7: Range-for over a sub-range

Using range-base loops, you can loop over a sub-part of a given container or other range by generating a proxy object that qualifies for range-based for loops.

```
template<class Iterator, class Sentinel=Iterator>
struct range_t {
  Iterator b;
  Sentinel e;
  Iterator begin() const { return b; }
  Sentinel end() const { return e; }
  bool empty() const { return begin()==end(); }
  range_t without_front( std::size_t count=1 ) const {
    if (std::is_same< std::random_access_iterator_tag, typename
std::iterator_traits<Iterator>::iterator_category >{} ) {
      count = (std::min)(std::size_t(std::distance(b,e)), count);
    }
    return {std::next(b, count), e};
  }
  range_t without_back( std::size_t count=1 ) const {
    if (std::is_same< std::random_access_iterator_tag, typename
std::iterator_traits<Iterator>::iterator_category >{} ) {
      count = (std::min)(std::size_t(std::distance(b,e)), count);
    }
    return {b, std::prev(e, count)};
  }
};

template<class Iterator, class Sentinel>
range_t<Iterator, Sentinel> range( Iterator b, Sentinal e ) {
  return {b,e};
}
template<class Iterable>
auto range( Iterable& r ) {
  using std::begin; using std::end;
  return range(begin(r),end(r));
}

template<class C>
auto except_first( C& c ) {
  auto r = range(c);
  if (r.empty()) return r;
  return r.without_front();
}
```

now we can do:

```
std::vector<int> v = {1,2,3,4};

for (auto i : except_first(v))
  std::cout << i << '\n';
```

and print out

```
2
```

```
3
4
```

Be aware that intermediate objects generated in the `for(:range_expression)` part of the `for` loop will have expired by the time the `for` loop starts.

Chapter 12: File I/O

C++ file I/O is done via *streams*. The key abstractions are:

`std::istream` for reading text.

`std::ostream` for writing text.

`std::streambuf` for reading or writing characters.

Formatted input uses `operator>>`.

Formatted output uses `operator<<`.

Streams use `std::locale`, e.g., for details of the formatting and for translation between external encodings and the internal encoding.

More on streams: <iostream> Library

Section 12.1: Writing to a file

There are several ways to write to a file. The easiest way is to use an output file stream (`ofstream`) together with the stream insertion operator (`<<`):

```
std::ofstream os("foo.txt");
if(os.is_open()){
    os << "Hello World!";
}
```

Instead of `<<`, you can also use the output file stream's member function `write()`:

```
std::ofstream os("foo.txt");
if(os.is_open()){
    char data[] = "Foo";

    // Writes 3 characters from data -> "Foo".
    os.write(data, 3);
}
```

After writing to a stream, you should always check if error state flag `badbit` has been set, as it indicates whether the operation failed or not. This can be done by calling the output file stream's member function `bad()`:

```
os << "Hello Badbit!"; // This operation might fail for any reason.
if (os.bad())
    // Failed to write!
```

Section 12.2: Opening a file

Opening a file is done in the same way for all 3 file streams (`ifstream`, `ofstream`, and `fstream`).

You can open the file directly in the constructor:

```
std::ifstream ifs("foo.txt");  // ifstream: Opens file "foo.txt" for reading only.

std::ofstream ofs("foo.txt");  // ofstream: Opens file "foo.txt" for writing only.
```

```
std::fstream iofs("foo.txt");  // fstream:  Opens file "foo.txt" for reading and writing.
```

Alternatively, you can use the file stream's member function `open()`:

```
std::ifstream ifs;
ifs.open("bar.txt");           // ifstream: Opens file "bar.txt" for reading only.

std::ofstream ofs;
ofs.open("bar.txt");           // ofstream: Opens file "bar.txt" for writing only.

std::fstream iofs;
iofs.open("bar.txt");          // fstream:  Opens file "bar.txt" for reading and writing.
```

You should **always** check if a file has been opened successfully (even when writing). Failures can include: the file doesn't exist, file hasn't the right access rights, file is already in use, disk errors occurred, drive disconnected ... Checking can be done as follows:

```
// Try to read the file 'foo.txt'.
std::ifstream ifs("fooo.txt");  // Note the typo; the file can't be opened.

// Check if the file has been opened successfully.
if (!ifs.is_open()) {
    // The file hasn't been opened; take appropriate actions here.
    throw CustomException(ifs, "File could not be opened");
}
```

When file path contains backslashes (for example, on Windows system) you should properly escape them:

```
// Open the file 'c:\\folder\\foo.txt' on Windows.
std::ifstream ifs("c:\\\\folder\\\\foo.txt"); // using escaped backslashes
```

Version ≥ C++11

or use raw literal:

```
// Open the file 'c:\\folder\\foo.txt' on Windows.
std::ifstream ifs(R"(c:\\folder\\foo.txt)"); // using raw literal
```

or use forward slashes instead:

```
// Open the file 'c:\\folder\\foo.txt' on Windows.
std::ifstream ifs("c:/folder/foo.txt");
```

Version ≥ C++11

If you want to open file with non-ASCII characters in path on Windows currently you can use **non-standard** wide character path argument:

```
// Open the file 'пример\\foo.txt' on Windows.
std::ifstream ifs(LR"(пример\\foo.txt)"); // using wide characters with raw literal
```

Section 12.3: Reading from a file

There are several ways to read data from a file.

If you know how the data is formatted, you can use the stream extraction operator (`>>`). Let's assume you have a file named *foo.txt* which contains the following data:

```
John Doe 25 4 6 1987
Jane Doe 15 5 24 1976
```

Then you can use the following code to read that data from the file:

```
// Define variables.
std::ifstream is("foo.txt");
std::string firstname, lastname;
int age, bmonth, bday, byear;

// Extract firstname, lastname, age, bday month, bday day, and bday year in that order.
// Note: '>>' returns false if it reached EOF (end of file) or if the input data doesn't
// correspond to the type of the input variable (for example, the string "foo" can't be
// extracted into an 'int' variable).
while (is >> firstname >> lastname >> age >> bmonth >> bday >> byear)
    // Process the data that has been read.
```

The stream extraction operator `>>` extracts every character and stops if it finds a character that can't be stored or if it is a special character:

- For string types, the operator stops at a whitespace () or at a newline (`\n`).
- For numbers, the operator stops at a non-number character.

This means that the following version of the file *foo.txt* will also be successfully read by the previous code:

```
John
Doe 25
4 6 1987


Jane
Doe
15 5
24
1976
```

The stream extraction operator `>>` always returns the stream given to it. Therefore, multiple operators can be chained together in order to read data consecutively. However, a stream can also be used as a Boolean expression (as shown in the `while` loop in the previous code). This is because the stream classes have a conversion operator for the type `bool`. This `bool()` operator will return `true` as long as the stream has no errors. If a stream goes into an error state (for example, because no more data can be extracted), then the `bool()` operator will return `false`. Therefore, the `while` loop in the previous code will be exited after the input file has been read to its end.

If you wish to read an entire file as a string, you may use the following code:

```
// Opens 'foo.txt'.
std::ifstream is("foo.txt");
std::string whole_file;

// Sets position to the end of the file.
is.seekg(0, std::ios::end);

// Reserves memory for the file.
whole_file.reserve(is.tellg());

// Sets position to the start of the file.
is.seekg(0, std::ios::beg);
```

```
// Sets contents of 'whole_file' to all characters in the file.
whole_file.assign(std::istreambuf_iterator<char>(is),
  std::istreambuf_iterator<char>());
```

This code reserves space for the `string` in order to cut down on unneeded memory allocations.

If you want to read a file line by line, you can use the function `getline()`:

```
std::ifstream is("foo.txt");

// The function getline returns false if there are no more lines.
for (std::string str; std::getline(is, str);) {
    // Process the line that has been read.
}
```

If you want to read a fixed number of characters, you can use the stream's member function `read()`:

```
std::ifstream is("foo.txt");
char str[4];

// Read 4 characters from the file.
is.read(str, 4);
```

After executing a read command, you should always check if the error state flag `failbit` has been set, as it indicates whether the operation failed or not. This can be done by calling the file stream's member function `fail()`:

```
is.read(str, 4); // This operation might fail for any reason.

if (is.fail())
    // Failed to read!
```

Section 12.4: Opening modes

When creating a file stream, you can specify an opening mode. An opening mode is basically a setting to control how the stream opens the file.

(All modes can be found in the `std::ios` namespace.)

An opening mode can be provided as second parameter to the constructor of a file stream or to its `open()` member function:

```
std::ofstream os("foo.txt", std::ios::out | std::ios::trunc);

std::ifstream is;
is.open("foo.txt", std::ios::in | std::ios::binary);
```

It is to be noted that you have to set `ios::in` or `ios::out` if you want to set other flags as they are not implicitly set by the iostream members although they have a correct default value.

If you don't specify an opening mode, then the following default modes are used:

- `ifstream` - `in`
- `ofstream` - `out`
- `fstream` - `in` and `out`

The file opening modes that you may specify by design are:

Mode	Meaning	For	Description
app	append	Output	Appends data at the end of the file.
binary	binary	Input/Output	Input and output is done in binary.
in	input	Input	Opens the file for reading.
out	output	Output	Opens the file for writing.
trunc	truncate	Input/Output	Removes contents of the file when opening.
ate	at end	Input	Goes to the end of the file when opening.

Note: Setting the `binary` mode lets the data be read/written exactly as-is; not setting it enables the translation of the newline `'\n'` character to/from a platform specific end of line sequence.

For example on Windows the end of line sequence is CRLF (`"\r\n"`).
Write: `"\n"` => `"\r\n"`
Read: `"\r\n"` => `"\n"`

Section 12.5: Reading an ASCII file into a std::string

```
std::ifstream f("file.txt");

if (f)
{
  std::stringstream buffer;
  buffer << f.rdbuf();
  f.close();

  // The content of "file.txt" is available in the string `buffer.str()`
}
```

The `rdbuf()` method returns a pointer to a `streambuf` that can be pushed into `buffer` via the `stringstream::operator<<` member function.

Another possibility (popularized in Effective STL by Scott Meyers) is:

```
std::ifstream f("file.txt");

if (f)
{
  std::string str((std::istreambuf_iterator<char>(f)),
                  std::istreambuf_iterator<char>());

  // Operations on `str`...
}
```

This is nice because requires little code (and allows reading a file directly into any STL container, not only strings) but can be slow for big files.

NOTE: the extra parentheses around the first argument to the string constructor are essential to prevent the *most vexing parse* problem.

Last but not least:

```
std::ifstream f("file.txt");

if (f)
{
  f.seekg(0, std::ios::end);
```

```
    const auto size = f.tellg();

    std::string str(size, ' ');
    f.seekg(0);
    f.read(&str[0], size);
    f.close();

    // Operations on `str`...
}
```

which is probably the fastest option (among the three proposed).

Section 12.6: Writing files with non-standard locale settings

If you need to write a file using different locale settings to the default, you can use `std::locale` and `std::basic_ios::imbue()` to do that for a specific file stream:

Guidance for use:

- You should always apply a local to a stream before opening the file.
- Once the stream has been imbued you should not change the locale.

Reasons for Restrictions: Imbuing a file stream with a locale has undefined behavior if the current locale is not state independent or not pointing at the beginning of the file.

UTF-8 streams (and others) are not state independent. Also a file stream with a UTF-8 locale may try and read the BOM marker from the file when it is opened; so just opening the file may read characters from the file and it will not be at the beginning.

```
#include <iostream>
#include <fstream>
#include <locale>

int main()
{
    std::cout << "User-preferred locale setting is "
              << std::locale("").name().c_str() << std::endl;

    // Write a floating-point value using the user's preferred locale.
    std::ofstream ofs1;
    ofs1.imbue(std::locale(""));
    ofs1.open("file1.txt");
    ofs1 << 78123.456 << std::endl;

    // Use a specific locale (names are system-dependent)
    std::ofstream ofs2;
    ofs2.imbue(std::locale("en_US.UTF-8"));
    ofs2.open("file2.txt");
    ofs2 << 78123.456 << std::endl;

    // Switch to the classic "C" locale
    std::ofstream ofs3;
    ofs3.imbue(std::locale::classic());
    ofs3.open("file3.txt");
    ofs3 << 78123.456 << std::endl;
}
```

Explicitly switching to the classic "C" locale is useful if your program uses a different default locale and you want to

ensure a fixed standard for reading and writing files. With a "C" preferred locale, the example writes

```
78,123.456
78,123.456
78123.456
```

If, for example, the preferred locale is German and hence uses a different number format, the example writes

```
78 123,456
78,123.456
78123.456
```

(note the decimal comma in the first line).

Section 12.7: Checking end of file inside a loop condition, bad practice?

`eof` returns `true` only **after** reading the end of file. It does NOT indicate that the next read will be the end of stream.

```
while (!f.eof())
{
  // Everything is OK

  f >> buffer;

  // What if *only* now the eof / fail bit is set?

  /* Use `buffer` */
}
```

You could correctly write:

```
while (!f.eof())
{
  f >> buffer >> std::ws;

  if (f.fail())
    break;

  /* Use `buffer` */
}
```

but

```
while (f >> buffer)
{
  /* Use `buffer` */
}
```

is simpler and less error prone.

Further references:

- `std::ws`: discards leading whitespace from an input stream
- `std::basic_ios::fail`: returns `true` if an error has occurred on the associated stream

Section 12.8: Flushing a stream

File streams are buffered by default, as are many other types of streams. This means that writes to the stream may not cause the underlying file to change immediately. In oder to force all buffered writes to take place immediately, you can *flush* the stream. You can do this either directly by invoking the `flush()` method or through the `std::flush` stream manipulator:

```
std::ofstream os("foo.txt");
os << "Hello World!" << std::flush;

char data[3] = "Foo";
os.write(data, 3);
os.flush();
```

There is a stream manipulator `std::endl` that combines writing a newline with flushing the stream:

```
// Both following lines do the same thing
os << "Hello World!\n" << std::flush;
os << "Hello world!" << std::endl;
```

Buffering can improve the performance of writing to a stream. Therefore, applications that do a lot of writing should avoid flushing unnecessarily. Contrary, if I/O is done infrequently, applications should consider flushing frequently in order to avoid data getting stuck in the stream object.

Section 12.9: Reading a file into a container

In the example below we use `std::string` and `operator>>` to read items from the file.

```
    std::ifstream file("file3.txt");

    std::vector<std::string>  v;

    std::string s;
    while(file >> s) // keep reading until we run out
    {
        v.push_back(s);
    }
```

In the above example we are simply iterating through the file reading one "item" at a time using `operator>>`. This same affect can be achieved using the `std::istream_iterator` which is an input iterator that reads one "item" at a time from the stream. Also most containers can be constructed using two iterators so we can simplify the above code to:

```
    std::ifstream file("file3.txt");

    std::vector<std::string>  v(std::istream_iterator<std::string>{file},
                                std::istream_iterator<std::string>{});
```

We can extend this to read any object types we like by simply specifying the object we want to read as the template parameter to the `std::istream_iterator`. Thus we can simply extend the above to read lines (rather than words) like this:

```
// Unfortunately there is  no built in type that reads line using >>
// So here we build a simple helper class to do it. That will convert
// back to a string when used in string context.
struct Line
```

```
{
    // Store data here
    std::string data;
    // Convert object to string
    operator std::string const&() const {return data;}
    // Read a line from a stream.
    friend std::istream& operator>>(std::istream& stream, Line& line)
    {
        return std::getline(stream, line.data);
    }
};


    std::ifstream file("file3.txt");

    // Read the lines of a file into a container.
    std::vector<std::string>  v(std::istream_iterator<Line>{file},
                                std::istream_iterator<Line>{});
```

Section 12.10: Copying a file

```
std::ifstream  src("source_filename", std::ios::binary);
std::ofstream  dst("dest_filename",   std::ios::binary);
dst << src.rdbuf();
```

Version ≥ C++17

With C++17 the standard way to copy a file is including the **<filesystem>** header and using copy_file:

```
std::fileystem::copy_file("source_filename", "dest_filename");
```

The filesystem library was originally developed as `boost.filesystem` and finally merged to ISO C++ as of C++17.

Section 12.11: Closing a file

Explicitly closing a file is rarely necessary in C++, as a file stream will automatically close its associated file in its destructor. However, you should try to limit the lifetime of a file stream object, so that it does not keep the file handle open longer than necessary. For example, this can be done by putting all file operations into an own scope (`{}`):

```
std::string const prepared_data = prepare_data();
{
    // Open a file for writing.
    std::ofstream output("foo.txt");

    // Write data.
    output << prepared_data;
}  // The ofstream will go out of scope here.
   // Its destructor will take care of closing the file properly.
```

Calling `close()` explicitly is only necessary if you want to reuse the same `fstream` object later, but don't want to keep the file open in between:

```
// Open the file "foo.txt" for the first time.
std::ofstream output("foo.txt");

// Get some data to write from somewhere.
std::string const prepared_data = prepare_data();
```

```
// Write data to the file "foo.txt".
output << prepared_data;

// Close the file "foo.txt".
output.close();

// Preparing data might take a long time. Therefore, we don't open the output file stream
// before we actually can write some data to it.
std::string const more_prepared_data = prepare_complex_data();

// Open the file "foo.txt" for the second time once we are ready for writing.
output.open("foo.txt");

// Write the data to the file "foo.txt".
output << more_prepared_data;

// Close the file "foo.txt" once again.
output.close();
```

Section 12.12: Reading a `struct` from a formatted text file

Version ≥ C++11

```
struct info_type
{
    std::string name;
    int age;
    float height;

    // we define an overload of operator>> as a friend function which
    // gives in privileged access to private data members
    friend std::istream& operator>>(std::istream& is, info_type& info)
    {
        // skip whitespace
        is >> std::ws;
        std::getline(is, info.name);
        is >> info.age;
        is >> info.height;
        return is;
    }
};

void func4()
{
    auto file = std::ifstream("file4.txt");

    std::vector<info_type> v;

    for(info_type info; file >> info;) // keep reading until we run out
    {
        // we only get here if the read succeeded
        v.push_back(info);
    }

    for(auto const& info: v)
    {
        std::cout << "  name: " << info.name << '\n';
        std::cout << "   age: " << info.age << " years" << '\n';
        std::cout << "height: " << info.height << "lbs" << '\n';
        std::cout << '\n';
    }
}
```

file4.txt

```
Wogger Wabbit
2
6.2
Bilbo Baggins
111
81.3
Mary Poppins
29
154.8
```

Output:

```
name: Wogger Wabbit
 age: 2 years
height: 6.2lbs

name: Bilbo Baggins
 age: 111 years
height: 81.3lbs

name: Mary Poppins
 age: 29 years
height: 154.8lbs
```

Chapter 13: C++ Streams

Section 13.1: String streams

`std::ostringstream` is a class whose objects look like an output stream (that is, you can write to them via `operator<<`), but actually store the writing results, and provide them in the form of a stream.

Consider the following short code:

```
#include <sstream>
#include <string>


using namespace std;

int main()
{
    ostringstream ss;
    ss << "the answer to everything is " << 42;
    const string result = ss.str();
}
```

The line

```
ostringstream ss;
```

creates such an object. This object is first manipulated like a regular stream:

```
ss << "the answer to everything is " << 42;
```

Following that, though, the resulting stream can be obtained like this:

```
const string result = ss.str();
```

(the string `result` will be equal to `"the answer to everything is 42"`).

This is mainly useful when we have a class for which stream serialization has been defined, and for which we want a string form. For example, suppose we have some class

```
class foo
{
    // All sort of stuff here.
};

ostream &operator<<(ostream &os, const foo &f);
```

To get the string representation of a `foo` object,

```
foo f;
```

we could use

```
ostringstream ss;
ss << f;
const string result = ss.str();
```

Then `result` contains the string representation of the `foo` object.

Section 13.2: Printing collections with iostream

Basic printing

`std::ostream_iterator` allows to print contents of an STL container to any output stream without explicit loops. The second argument of `std::ostream_iterator` constructor sets the delimiter. For example, the following code:

```
std::vector<int> v = {1,2,3,4};
std::copy(v.begin(), v.end(), std::ostream_iterator<int>(std::cout, " ! "));
```

will print

```
1 ! 2 ! 3 ! 4 !
```

Implicit type cast

`std::ostream_iterator` allows to cast container's content type implicitly. For example, let's tune `std::cout` to print floating-point values with 3 digits after decimal point:

```
std::cout << std::setprecision(3);
std::fixed(std::cout);
```

and instantiate `std::ostream_iterator` with `float`, while the contained values remain `int`:

```
std::vector<int> v = {1,2,3,4};
std::copy(v.begin(), v.end(), std::ostream_iterator<float>(std::cout, " ! "));
```

so the code above yields

```
1.000 ! 2.000 ! 3.000 ! 4.000 !
```

despite `std::vector` holds `int`s.

Generation and transformation

`std::generate`, `std::generate_n` and `std::transform` functions provide a very powerful tool for on-the-fly data manipulation. For example, having a vector:

```
std::vector<int> v = {1,2,3,4,8,16};
```

we can easily print boolean value of "x is even" statement for each element:

```
std::boolalpha(std::cout); // print booleans alphabetically
std::transform(v.begin(), v.end(), std::ostream_iterator<bool>(std::cout, " "),
[](int val) {
    return (val % 2) == 0;
});
```

or print the squared element:

```
std::transform(v.begin(), v.end(), std::ostream_iterator<int>(std::cout, " "),
[](int val) {
    return val * val;
```

```
});
```

Printing N space-delimited random numbers:

```
const int N = 10;
std::generate_n(std::ostream_iterator<int>(std::cout, " "), N, std::rand);
```

Arrays

As in the section about reading text files, almost all these considerations may be applied to native arrays. For example, let's print squared values from a native array:

```
int v[] = {1,2,3,4,8,16};
std::transform(v, std::end(v), std::ostream_iterator<int>(std::cout, " "),
[](int val) {
    return val * val;
});
```

Chapter 14: Stream manipulators

Manipulators are special helper functions that help controlling input and output streams using `operator >>` or `operator <<`.

They all can be included by `#include <iomanip>`.

Section 14.1: Stream manipulators

`std::boolalpha` and `std::noboolalpha` - switch between textual and numeric representation of booleans.

```
std::cout << std::boolalpha << 1;
// Output: true

std::cout << std::noboolalpha << false;
// Output: 0

bool boolValue;
std::cin >> std::boolalpha >> boolValue;
std::cout << "Value \"" << std::boolalpha << boolValue
          << "\" was parsed as " << std::noboolalpha << boolValue;
// Input: true
// Output: Value "true" was parsed as 0
```

`std::showbase` and `std::noshowbase` - control whether prefix indicating numeric base is used.

`std::dec` (decimal), `std::hex` (hexadecimal) and `std::oct` (octal) - are used for changing base for integers.

```
#include <sstream>

std::cout << std::dec << 29 << ' - '
          << std::hex << 29 << ' - '
          << std::showbase << std::oct << 29 << ' - '
          << std::noshowbase << 29  '\n';
int number;
std::istringstream("3B") >> std::hex >> number;
std::cout << std::dec << 10;
// Output: 22 - 1D - 35 - 035
// 59
```

Default values are `std::ios_base::noshowbase` and `std::ios_base::dec`.

If you want to see more about `std::istringstream` check out the <sstream> header.

`std::uppercase` and `std::nouppercase` - control whether uppercase characters are used in floating-point and hexadecimal integer output. Have no effect on input streams.

```
std::cout << std::hex << std::showbase
              << "0x2a with nouppercase: " << std::nouppercase << 0x2a << '\n'
              << "1e-10 with uppercase: " << std::uppercase << 1e-10 << '\n'
}
// Output: 0x2a with nouppercase: 0x2a
// 1e-10 with uppercase: 1E-10
```

Default is `std::nouppercase`.

`std::setw(n)` - changes the width of the next input/output field to exactly `n`.

The width property `n` is resetting to `0` when some functions are called (full list is here).

```
std::cout << "no setw:" << 51 << '\n'
          << "setw(7): " << std::setw(7) << 51 << '\n'
          << "setw(7), more output: " << 13
          << std::setw(7) << std::setfill('*') << 67 << ' ' << 94 << '\n';

char* input = "Hello, world!";
char arr[10];
std::cin >> std::setw(6) >> arr;
std::cout << "Input from \"Hello, world!\" with setw(6) gave \"" << arr << "\"\n";

// Output: 51
// setw(7):      51
// setw(7), more output: 13*****67 94

// Input: Hello, world!
// Output: Input from "Hello, world!" with setw(6) gave "Hello"
```

Default is `std::setw(0)`.

`std::left`, `std::right` and `std::internal` - modify the default position of the fill characters by setting `std::ios_base::adjustfield` to `std::ios_base::left`, `std::ios_base::right` and `std::ios_base::internal` correspondingly. `std::left` and `std::right` apply to any output, `std::internal` - for integer, floating-point and monetary output. Have no effect on input streams.

```
#include <locale>
...

std::cout.imbue(std::locale("en_US.utf8"));

std::cout << std::left << std::showbase << std::setfill('*')
          << "flt: " << std::setw(15) << -9.87  << '\n'
          << "hex: " << std::setw(15) << 41 << '\n'
          << "  $: " << std::setw(15) << std::put_money(367, false) << '\n'
          << "usd: " << std::setw(15) << std::put_money(367, true) << '\n'
          << "usd: " << std::setw(15)
          << std::setfill(' ') << std::put_money(367, false) << '\n';
// Output:
// flt: -9.87**********
// hex: 41*************
//   $: $3.67**********
// usd: USD *3.67******
// usd: $3.67

std::cout << std::internal << std::showbase << std::setfill('*')
          << "flt: " << std::setw(15) << -9.87  << '\n'
          << "hex: " << std::setw(15) << 41 << '\n'
          << "  $: " << std::setw(15) << std::put_money(367, false) << '\n'
          << "usd: " << std::setw(15) << std::put_money(367, true) << '\n'
          << "usd: " << std::setw(15)
          << std::setfill(' ') << std::put_money(367, true) << '\n';
```

```
// Output:
// flt: -**********9.87
// hex: *************41
//   $: $3.67**********
// usd: USD *******3.67
// usd: USD        3.67

std::cout << std::right << std::showbase << std::setfill('*')
          << "flt: " << std::setw(15) << -9.87  << '\n'
          << "hex: " << std::setw(15) << 41 << '\n'
          << "  $: " << std::setw(15) << std::put_money(367, false) << '\n'
          << "usd: " << std::setw(15) << std::put_money(367, true) << '\n'
          << "usd: " << std::setw(15)
          << std::setfill(' ') << std::put_money(367, true) << '\n';
// Output:
// flt: **********-9.87
// hex: *************41
//   $: **********$3.67
// usd: ******USD *3.67
// usd:       USD  3.67
```

Default is `std::left`.

`std::fixed`, `std::scientific`, `std::hexfloat` [C++11] and `std::defaultfloat` [C++11] - change formatting for floating-point input/output.

`std::fixed` sets the `std::ios_base::floatfield` to `std::ios_base::fixed`,
`std::scientific` - to `std::ios_base::scientific`,
`std::hexfloat` - to `std::ios_base::fixed | std::ios_base::scientific` and
`std::defaultfloat` - to `std::ios_base::fmtflags(0)`.

`fmtflags`

```
#include <sstream>
...

std::cout << '\n'
          << "The number 0.07 in fixed:      " << std::fixed << 0.01 << '\n'
          << "The number 0.07 in scientific: " << std::scientific << 0.01 << '\n'
          << "The number 0.07 in hexfloat:   " << std::hexfloat << 0.01 << '\n'
          << "The number 0.07 in default:    " << std::defaultfloat << 0.01 << '\n';

double f;
std::istringstream is("0x1P-1022");
double f = std::strtod(is.str().c_str(), NULL);
std::cout << "Parsing 0x1P-1022 as hex gives " << f << '\n';

// Output:
// The number 0.01 in fixed:      0.070000
// The number 0.01 in scientific: 7.000000e-02
// The number 0.01 in hexfloat:   0x1.1eb851eb851ecp-4
// The number 0.01 in default:    0.07
// Parsing 0x1P-1022 as hex gives 2.22507e-308
```

Default is `std::ios_base::fmtflags(0)`.

There is a **bug** on some compilers which causes

```
double f;
std::istringstream("0x1P-1022") >> std::hexfloat >> f;
std::cout << "Parsing 0x1P-1022 as hex gives " << f << '\n';
// Output: Parsing 0x1P-1022 as hex gives 0
```

`std::showpoint` and `std::noshowpoint` - control whether decimal point is always included in floating-point representation. Have no effect on input streams.

```
std::cout << "7.0 with showpoint: " << std::showpoint << 7.0 << '\n'
          << "7.0 with noshowpoint: " << std::noshowpoint << 7.0 << '\n';
// Output: 1.0 with showpoint: 7.00000
// 1.0 with noshowpoint: 7
```

Default is `std::showpoint`.

`std::showpos` and `std::noshowpos` - control displaying of the `+` sign in *non-negative* output. Have no effect on input streams.

```
std::cout << "With showpos: " << std::showpos
          << 0 << ' ' << -2.718 << ' ' << 17 << '\n'
          << "Without showpos: " << std::noshowpos
          << 0 << ' ' << -2.718 << ' ' << 17 << '\n';
// Output: With showpos: +0 -2.718 +17
// Without showpos: 0 -2.718 17
```

Default if `std::noshowpos`.

`std::unitbuf`, `std::nounitbuf` - control flushing output stream after every operation. Have no effect on input stream. `std::unitbuf` causes flushing.

`std::setbase(base)` - sets the numeric base of the stream.

`std::setbase(8)` equals to setting `std::ios_base::basefield` to `std::ios_base::oct`,
`std::setbase(16)` - to `std::ios_base::hex`,
`std::setbase(10)` - to `std::ios_base::dec`.

If `base` is other then `8`, `10` or `16` then `std::ios_base::basefield` is setting to `std::ios_base::fmtflags(0)`. It means decimal output and prefix-dependent input.

As default `std::ios_base::basefield` is `std::ios_base::dec` then by default `std::setbase(10)`.

`std::setprecision(n)` - changes floating-point precision.

```
#include <cmath>
#include <limits>
...

typedef std::numeric_limits<long double> ld;
```

```
const long double pi = std::acos(-1.L);

std::cout << '\n'
          << "default precision (6):   pi: " << pi << '\n'
          << "                       10pi: " << 10 * pi << '\n'
          << "std::setprecision(4):  10pi: " << std::setprecision(4) << 10 * pi << '\n'
          << "                    10000pi: " << 10000 * pi << '\n'
          << "std::fixed:         10000pi: " << std::fixed << 10000 * pi << std::defaultfloat <<
'\n'
          << "std::setprecision(10):   pi: " << std::setprecision(10) << pi << '\n'
          << "max-1 radix precicion:   pi: " << std::setprecision(ld::digits - 1) << pi << '\n'
          << "max+1 radix precision:   pi: " << std::setprecision(ld::digits + 1) << pi << '\n'
          << "significant digits prec: pi: " << std::setprecision(ld::digits10) << pi << '\n';

// Output:
// default precision (6):   pi: 3.14159
//                        10pi: 31.4159
// std::setprecision(4):  10pi: 31.42
//                     10000pi: 3.142e+04
// std::fixed:         10000pi: 31415.9265
// std::setprecision(10):   pi: 3.141592654
// max-1 radix precicion:   pi: 3.14159265358979323851280895940618620443274267017841339111328125
// max+1 radix precision:   pi: 3.14159265358979323851280895940618620443274267017841339111328125
// significant digits prec: pi: 3.14159265358979324
```

Default is `std::setprecision(6)`.

`std::setiosflags(mask)` and `std::resetiosflags(mask)` - set and clear flags specified in `mask` of `std::ios_base::fmtflags` type.

```
#include <sstream>
...

std::istringstream in("10 010 10 010 10 010");
int num1, num2;

in >> std::oct >> num1 >> num2;
std::cout << "Parsing \"10 010\" with std::oct gives:   " << num1 << ' ' << num2 << '\n';
// Output: Parsing "10 010" with std::oct gives:   8 8

in >> std::dec >> num1 >> num2;
std::cout << "Parsing \"10 010\" with std::dec gives:   " << num1 << ' ' << num2 << '\n';
// Output: Parsing "10 010" with std::oct gives:   10 10

in >> std::resetiosflags(std::ios_base::basefield) >> num1 >> num2;
std::cout << "Parsing \"10 010\" with autodetect gives: " << num1 << ' ' << num2 << '\n';
// Parsing "10 010" with autodetect gives: 10 8

std::cout << std::setiosflags(std::ios_base::hex |
                              std::ios_base::uppercase |
                              std::ios_base::showbase) << 42 << '\n';
// Output: OX2A
```

`std::skipws` and `std::noskipws` - control skipping of leading whitespace by the formatted input functions. Have no effect on output streams.

```
#include <sstream>
```

```
...

char c1, c2, c3;
std::istringstream("a b c") >> c1 >> c2 >> c3;
std::cout << "Default  behavior:  c1 = " << c1 << "  c2 = " << c2 << "  c3 = " << c3 << '\n';

std::istringstream("a b c") >> std::noskipws >> c1 >> c2 >> c3;
std::cout << "noskipws behavior:  c1 = " << c1 << "  c2 = " << c2 << "  c3 = " << c3 << '\n';
// Output: Default  behavior:  c1 = a  c2 = b  c3 = c
// noskipws behavior:  c1 = a  c2 =    c3 = b
```

Default is `std::ios_base::skipws`.

`std::quoted(s[, delim[, escape]])` [C++14] - inserts or extracts quoted strings with embedded spaces.

`s` - the string to insert or extract.
`delim` - the character to use as the delimiter, `"` by default.
`escape` - the character to use as the escape character, `\` by default.

```
#include <sstream>
...

std::stringstream ss;
std::string in = "String with spaces, and embedded \"quotes\" too";
std::string out;

ss << std::quoted(in);
std::cout << "read in     [" << in << "]\n"
          << "stored as   [" << ss.str() << "]\n";

ss >> std::quoted(out);
std::cout << "written out [" << out << "]\n";
// Output:
// read in     [String with spaces, and embedded "quotes" too]
// stored as   ["String with spaces, and embedded \"quotes\" too"]
// written out [String with spaces, and embedded "quotes" too]
```

For more information see the link above.

Section 14.2: Output stream manipulators

`std::ends` - inserts a null character `'\0'` to output stream. More formally this manipulator's declaration looks like

```
template <class charT, class traits>
std::basic_ostream<charT, traits>& ends(std::basic_ostream<charT, traits>& os);
```

and this manipulator places character by calling `os.put(charT())` when used in an expression
`os << std::ends;`

`std::endl` and `std::flush` both flush output stream `out` by calling `out.flush()`. It causes immediately producing output. But `std::endl` inserts end of line `'\n'` symbol before flushing.

```
std::cout << "First line." << std::endl << "Second line. " << std::flush
          << "Still second line.";
```

```
// Output: First line.
// Second line. Still second line.
```

std::setfill(c) - changes the fill character to c. Often used with std::setw.

```
std::cout << "\nDefault fill: " << std::setw(10) << 79 << '\n'
          << "setfill('#'): " << std::setfill('#')
          << std::setw(10) << 42 << '\n';
// Output:
// Default fill:         79
// setfill('#'): ########79
```

std::put_money(mon[, intl]) [C++11]. In an expression out << std::put_money(mon, intl), converts the monetary value mon (of long double or std::basic_string type) to its character representation as specified by the std::money_put facet of the locale currently imbued in out. Use international currency strings if intl is true, use currency symbols otherwise.

```
long double money = 123.45;
// or std::string money = "123.45";

std::cout.imbue(std::locale("en_US.utf8"));
std::cout << std::showbase << "en_US: " << std::put_money(money)
          << " or " << std::put_money(money, true) << '\n';
// Output: en_US: $1.23 or USD  1.23

std::cout.imbue(std::locale("ru_RU.utf8"));
std::cout << "ru_RU: " << std::put_money(money)
          << " or " << std::put_money(money, true) << '\n';
// Output: ru_RU: 1.23 руб or 1.23 RUB

std::cout.imbue(std::locale("ja_JP.utf8"));
std::cout << "ja_JP: " << std::put_money(money)
          << " or " << std::put_money(money, true) << '\n';
// Output: ja_JP: ￥123 or JPY  123
```

std::put_time(tmb, fmt) [C++11] - formats and outputs a date/time value to std::tm according to the specified format fmt.

tmb - pointer to the calendar time structure const std::tm* as obtained from localtime() or gmtime().
fmt - pointer to a null-terminated string const CharT* specifying the format of conversion.

```
#include <ctime>
...

std::time_t t = std::time(nullptr);
std::tm tm = *std::localtime(&t);

std::cout.imbue(std::locale("ru_RU.utf8"));
std::cout << "\nru_RU: " << std::put_time(&tm, "%c %Z") << '\n';
// Possible output:
// ru_RU: Вт 04 июл 2017 15:08:35 UTC
```

For more information see the link above.

Section 14.3: Input stream manipulators

`std::ws` - consumes leading whitespaces in input stream. It different from `std::skipws`.

```
#include <sstream>
...

std::string str;
std::istringstream("  \v\n\r\t    Wow!There   is no whitespaces!") >> std::ws >> str;
std::cout << str;
// Output: Wow!There   is no whitespaces!
```

`std::get_money(mon[, intl])` [C++11]. In an expression `in >> std::get_money(mon, intl)` parses the character input as a monetary value, as specified by the `std::money_get` facet of the locale currently imbued in `in`, and stores the value in `mon` (of `long double` or `std::basic_string` type). Manipulator expects *required* international currency strings if `intl` is `true`, expects *optional* currency symbols otherwise.

```
#include <sstream>
#include <locale>
...

std::istringstream in("$1,234.56 2.22 USD  3.33");
long double v1, v2;
std::string v3;

in.imbue(std::locale("en_US.UTF-8"));
in >> std::get_money(v1) >> std::get_money(v2) >> std::get_money(v3, true);
if (in) {
    std::cout << std::quoted(in.str()) << " parsed as: "
              << v1 << ", " << v2 << ", " << v3 << '\n';
}
// Output:
// "$1,234.56 2.22 USD  3.33" parsed as: 123456, 222, 333
```

`std::get_time(tmb, fmt)` [C++11] - parses a date/time value stored in `tmb` of specified format `fmt`.

`tmb` - valid pointer to the `const std::tm*` object where the result will be stored.
`fmt` - pointer to a null-terminated string `const CharT*` specifying the conversion format.

```
#include <sstream>
#include <locale>
...

std::tm t = {};
std::istringstream ss("2011-Februar-18 23:12:34");

ss.imbue(std::locale("de_DE.utf-8"));
ss >> std::get_time(&t, "%Y-%b-%d %H:%M:%S");
if (ss.fail()) {
    std::cout << "Parse failed\n";
}
else {
    std::cout << std::put_time(&t, "%c") << '\n';
}
// Possible output:
// Sun Feb 18 23:12:34 2011
```

For more information see the link above.

Chapter 15: Flow Control

Section 15.1: case

Introduces a case label of a switch statement. The operand must be a constant expression and match the switch condition in type. When the switch statement is executed, it will jump to the case label with operand equal to the condition, if any.

```
char c = getchar();
bool confirmed;
switch (c) {
  case 'y':
    confirmed = true;
    break;
  case 'n':
    confirmed = false;
    break;
  default:
    std::cout << "invalid response!\n";
    abort();
}
```

Section 15.2: switch

According to the C++ standard,

> The `switch` statement causes control to be transferred to one of several statements depending on the value of a condition.

The keyword `switch` is followed by a parenthesized condition and a block, which may contain `case` labels and an optional `default` label. When the switch statement is executed, control will be transferred either to a `case` label with a value matching that of the condition, if any, or to the `default` label, if any.

The condition must be an expression or a declaration, which has either integer or enumeration type, or a class type with a conversion function to integer or enumeration type.

```
char c = getchar();
bool confirmed;
switch (c) {
  case 'y':
    confirmed = true;
    break;
  case 'n':
    confirmed = false;
    break;
  default:
    std::cout << "invalid response!\n";
    abort();
}
```

Section 15.3: catch

The `catch` keyword introduces an exception handler, that is, a block into which control will be transferred when an exception of compatible type is thrown. The `catch` keyword is followed by a parenthesized *exception declaration*,

which is similar in form to a function parameter declaration: the parameter name may be omitted, and the ellipsis `...` is allowed, which matches any type. The exception handler will only handle the exception if its declaration is compatible with the type of the exception. For more details, see catching exceptions.

```
try {
    std::vector<int> v(N);
    // do something
} catch (const std::bad_alloc&) {
    std::cout << "failed to allocate memory for vector!" << std::endl;
} catch (const std::runtime_error& e) {
    std::cout << "runtime error: " << e.what() << std::endl;
} catch (...) {
    std::cout << "unexpected exception!" << std::endl;
    throw;
}
```

Section 15.4: throw

1. When `throw` occurs in an expression with an operand, its effect is to throw an exception, which is a copy of the operand.

    ```
    void print_asterisks(int count) {
        if (count < 0) {
            throw std::invalid_argument("count cannot be negative!");
        }
        while (count--) { putchar('*'); }
    }
    ```

2. When `throw` occurs in an expression without an operand, its effect is to rethrow the current exception. If there is no current exception, `std::terminate` is called.

    ```
    try {
        // something risky
    } catch (const std::bad_alloc&) {
        std::cerr << "out of memory" << std::endl;
    } catch (...) {
        std::cerr << "unexpected exception" << std::endl;
        // hope the caller knows how to handle this exception
        throw;
    }
    ```

3. When `throw` occurs in a function declarator, it introduces a dynamic exception specification, which lists the types of exceptions that the function is allowed to propagate.

    ```
    // this function might propagate a std::runtime_error,
    // but not, say, a std::logic_error
    void risky() throw(std::runtime_error);
    // this function can't propagate any exceptions
    void safe() throw();
    ```

    Dynamic exception specifications are deprecated as of C++11.

Note that the first two uses of `throw` listed above constitute expressions rather than statements. (The type of a throw expression is `void`.) This makes it possible to nest them within expressions, like so:

```
unsigned int predecessor(unsigned int x) {
    return (x > 0) ? (x - 1) : (throw std::invalid_argument("0 has no predecessor"));
}
```

Section 15.5: default

In a switch statement, introduces a label that will be jumped to if the condition's value is not equal to any of the case labels' values.

```
char c = getchar();
bool confirmed;
switch (c) {
  case 'y':
    confirmed = true;
    break;
  case 'n':
    confirmed = false;
    break;
  default:
    std::cout << "invalid response!\n";
    abort();
}
```

Version ≥ C++11

Defines a default constructor, copy constructor, move constructor, destructor, copy assignment operator, or move assignment operator to have its default behaviour.

```
class Base {
    // ...
    // we want to be able to delete derived classes through Base*,
    // but have the usual behaviour for Base's destructor.
    virtual ~Base() = default;
};
```

Section 15.6: try

The keyword `try` is followed by a block, or by a constructor initializer list and then a block (see here). The try block is followed by one or more catch blocks. If an exception propagates out of the try block, each of the corresponding catch blocks after the try block has the opportunity to handle the exception, if the types match.

```
std::vector<int> v(N);     // if an exception is thrown here,
                           // it will not be caught by the following catch block
try {
    std::vector<int> v(N); // if an exception is thrown here,
                           // it will be caught by the following catch block
    // do something with v
} catch (const std::bad_alloc&) {
    // handle bad_alloc exceptions from the try block
}
```

Section 15.7: if

Introduces an if statement. The keyword `if` must be followed by a parenthesized condition, which can be either an expression or a declaration. If the condition is truthy, the substatement after the condition will be executed.

```
int x;
```

```
std::cout << "Please enter a positive number." << std::endl;
std::cin >> x;
if (x <= 0) {
    std::cout << "You didn't enter a positive number!" << std::endl;
    abort();
}
```

Section 15.8: else

The first substatement of an if statement may be followed by the keyword `else`. The substatement after the `else` keyword will be executed when the condition is falsey (that is, when the first substatement is not executed).

```
int x;
std::cin >> x;
if (x%2 == 0) {
    std::cout << "The number is even\n";
} else {
    std::cout << "The number is odd\n";
}
```

Section 15.9: Conditional Structures: if, if..else

if and else:

it used to check whether the given expression returns true or false and acts as such:

```
if (condition) statement
```

the condition can be any valid C++ expression that returns something that be checked against truth/falsehood for example:

```
if (true) { /* code here */ }  // evaluate that true is true and execute the code in the brackets
if (false) { /* code here */ } // always skip the code since false is always false
```

the condition can be anything, a function, a variable, or a comparison for example

```
if(istrue()) { } // evaluate the function, if it returns true, the if will execute the code
if(isTrue(var)) { } //evaluate the return of the function after passing the argument var
if(a == b) { } // this will evaluate the return of the experssion (a==b) which will be true if
equal and false if unequal
if(a) { } //if a is a boolean type, it will evaluate for its value, if it's an integer, any non
zero value will be true,
```

if we want to check for a multiple expressions we can do it in two ways :

using binary operators :

```
if (a && b) { } // will be true only if both a and b are true (binary operators are outside the
scope here
if (a || b ) { } //true if a or b is true
```

using if/ifelse/else:

for a simple switch either if or else

```
if (a== "test") {
```

```
    //will execute if a is a string "test"
} else {
    // only if the first failed, will execute
}
```

for multiple choices :

```
if (a=='a') {
// if a is a char valued 'a'
} else if (a=='b') {
// if a is a char valued 'b'
} else if (a=='c') {
// if a is a char valued 'c'
} else {
//if a is none of the above
}
```

however it must be noted that you should use '**switch**' instead if your code checks for the same variable's value

Section 15.10: goto

Jumps to a labelled statement, which must be located in the current function.

```
bool f(int arg) {
    bool result = false;
    hWidget widget = get_widget(arg);
    if (!g()) {
        // we can't continue, but must do cleanup still
        goto end;
    }
    // ...
    result = true;
  end:
    release_widget(widget);
    return result;
}
```

Section 15.11: Jump statements : break, continue, goto, exit

The break instruction:

Using break we can leave a loop even if the condition for its end is not fulfilled. It can be used to end an infinite loop, or to force it to end before its natural end

The syntax is

```
break;
```

Example: we often use `break` in `switch` cases,ie once a case i switch is satisfied then the code block of that condition is executed .

```
switch(conditon){
case 1: block1;
case 2: block2;
case 3: block3;
default: blockdefault;
}
```

in this case if case 1 is satisfied then block 1 is executed , what we really want is only the block1 to be processed but instead once the block1 is processed remaining blocks,block2,block3 and blockdefault are also processed even though only case 1 was satified.To avoid this we use break at the end of each block like :

```
switch(condition){
case 1: block1;
        break;
case 2: block2;
        break;
case 3: block3;
        break;
default: blockdefault;
        break;
}
```

so only one block is processed and the control moves out of the switch loop.

break can also be used in other conditional and non conditional loops like `if`,`while`,`for` etc;

example:

```
if(condition1){
   ....
   if(condition2){
    .......
    break;
    }
 ...
}
```

The continue instruction:

The continue instruction causes the program to skip the rest of the loop in the present iteration as if the end of the statement block would have been reached, causing it to jump to the following iteration.

The syntax is

```
continue;
```

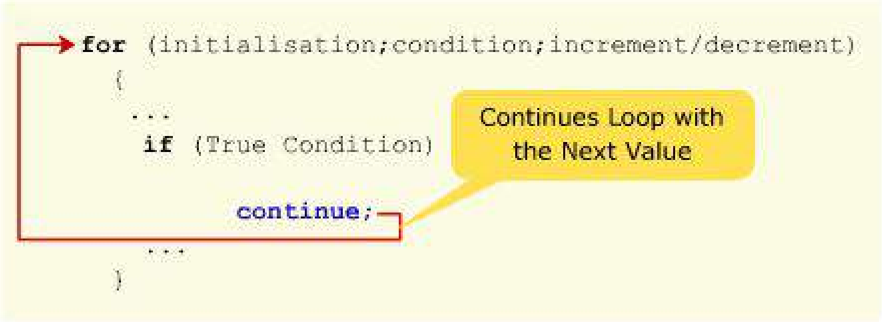
Example consider the following :

```
for(int i=0;i<10;i++){
if(i%2==0)
continue;
cout<<"\n @"<<i;
}
```

which produces the output:

```
 @1
 @3
 @5
 @7
 @9
```

i this code whenever the condition `i%2==0` is satisfied `continue` is processed,this causes the compiler to skip all the remaining code(printing @ and i) and increment/decrement statement of the loop gets executed.



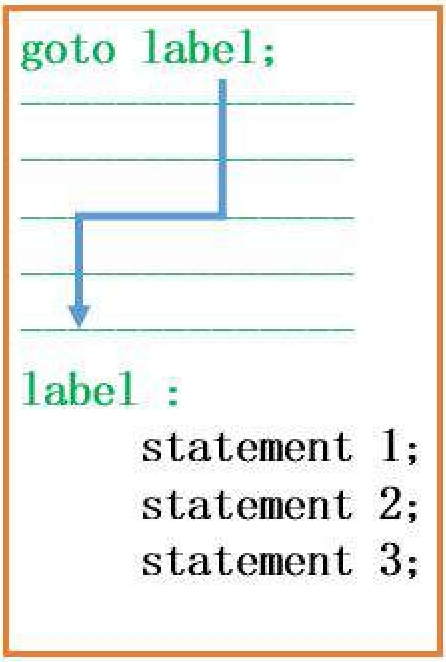
The goto instruction:

It allows making an absolute jump to another point in the program. You should use this feature carefully since its execution ignores any type of nesting limitation. The destination point is identified by a label, which is then used as an argument for the goto instruction. A label is made of a valid identifier followed by a colon (:)

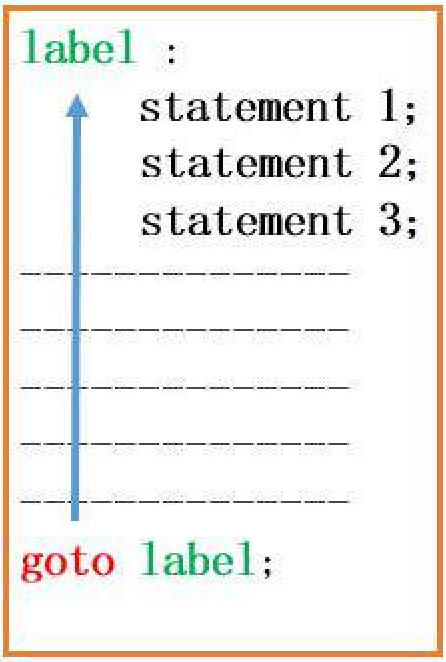
The syntax is

```
goto label;
..
.
label: statement;
```

Note: *Use of goto statement is highly discouraged because it makes difficult to trace the control flow of a program, making the program hard to understand and hard to modify.*



Forward Reference



Backward Reference

Example :

```
int num = 1;
STEP:
do{

    if( num%2==0 )
    {
        num = num + 1;
```

```
        goto STEP;
     }

   cout << "value of num : " << num << endl;
   num = num + 1;
 }while( num < 10 );
```

output :

```
value of num : 1
value of num : 3
value of num : 5
value of num : 7
value of num : 9
```

whenever the condition `num%2==0` is satisfied the goto sends the execution control to the beginning of the `do-while` loop.

The exit function:

`exit` is a function defined in `cstdlib`. The purpose of `exit` is to terminate the running program with an specific exit code. Its prototype is:

```
void exit (int exit code);
```

`cstdlib` defines the standard exit codes `EXIT_SUCCESS` and `EXIT_FAILURE`.

Section 15.12: return

Returns control from a function to its caller.

If `return` has an operand, the operand is converted to the function's return type, and the converted value is returned to the caller.

```
int f() {
    return 42;
}
int x = f(); // x is 42
int g() {
    return 3.14;
}
int y = g(); // y is 3
```

If `return` does not have an operand, the function must have `void` return type. As a special case, a `void`-returning function can also return an expression if the expression has type `void`.

```
void f(int x) {
    if (x < 0) return;
    std::cout << sqrt(x);
}
int g() { return 42; }
void h() {
    return f(); // calls f, then returns
    return g(); // ill-formed
}
```

When `main` returns, `std::exit` is implicitly called with the return value, and the value is thus returned to the

execution environment. (However, returning from `main` destroys automatic local variables, while calling `std::exit` directly does not.)

```
int main(int argc, char** argv) {
    if (argc < 2) {
        std::cout << "Missing argument\n";
        return EXIT_FAILURE; // equivalent to: exit(EXIT_FAILURE);
    }
}
```

Chapter 16: Metaprogramming

In C++ Metaprogramming refers to the use of macros or templates to generate code at compile-time.

In general, macros are frowned upon in this role and templates are preferred, although they are not as generic.

Template metaprogramming often makes use of compile-time computations, whether via templates or `constexpr` functions, to achieve its goals of generating code, however compile-time computations are not metaprogramming per se.

Section 16.1: Calculating Factorials

Factorials can be computed at compile-time using template metaprogramming techniques.

```
#include <iostream>

template<unsigned int n>
struct factorial
{
    enum
    {
        value = n * factorial<n - 1>::value
    };
};

template<>
struct factorial<0>
{
    enum { value = 1 };
};

int main()
{
    std::cout << factorial<7>::value << std::endl;    // prints "5040"
}
```

`factorial` is a struct, but in template metaprogramming it is treated as a template metafunction. By convention, template metafunctions are evaluated by checking a particular member, either `::type` for metafunctions that result in types, or `::value` for metafunctions that generate values.

In the above code, we evaluate the `factorial` metafunction by instantiating the template with the parameters we want to pass, and using `::value` to get the result of the evaluation.

The metafunction itself relies on recursively instantiating the same metafunction with smaller values. The `factorial<0>` specialization represents the terminating condition. Template metaprogramming has most of the restrictions of a functional programming language, so recursion is the primary "looping" construct.

Since template metafunctions execute at compile time, their results can be used in contexts that require compile-time values. For example:

```
int my_array[factorial<5>::value];
```

Automatic arrays must have a compile-time defined size. And the result of a metafunction is a compile-time constant, so it can be used here.

Limitation: Most of the compilers won't allow recursion depth beyond a limit. For example, `g++` compiler by default

limits recursion depeth to 256 levels. In case of `g++`, programmer can set recursion depth using `-ftemplate-depth-X` option.

Version ≥ C++11

Since C++11, the `std::integral_constant` template can be used for this kind of template computation:

```
#include <iostream>
#include <type_traits>

template<long long n>
struct factorial :
  std::integral_constant<long long, n * factorial<n - 1>::value> {};

template<>
struct factorial<0> :
  std::integral_constant<long long, 1> {};

int main()
{
    std::cout << factorial<7>::value << std::endl;    // prints "5040"
}
```

Additionally, `constexpr` functions become a cleaner alternative.

```
#include <iostream>

constexpr long long factorial(long long n)
{
  return (n == 0) ? 1 : n * factorial(n - 1);
}

int main()
{
  char test[factorial(3)];
  std::cout << factorial(7) << '\n';
}
```

The body of `factorial()` is written as a single statement because in C++11 `constexpr` functions can only use a quite limited subset of the language.

Version ≥ C++14

Since C++14, many restrictions for `constexpr` functions have been dropped and they can now be written much more conveniently:

```
constexpr long long factorial(long long n)
{
  if (n == 0)
    return 1;
  else
    return n * factorial(n - 1);
}
```

Or even:

```
constexpr long long factorial(int n)
{
  long long result = 1;
```

```
  for (int i = 1; i <= n; ++i) {
    result *= i;
  }
  return result;
}
```

Version ≥ C++17

Since c++17 one can use fold expression to calculate factorial:

```
#include <iostream>
#include <utility>

template <class T, T N, class I = std::make_integer_sequence<T, N>>
struct factorial;

template <class T, T N, T... Is>
struct factorial<T,N,std::index_sequence<T, Is...>> {
   static constexpr T value = (static_cast<T>(1) * ... * (Is + 1));
};

int main() {
   std::cout << factorial<int, 5>::value << std::endl;
}
```

Section 16.2: Iterating over a parameter pack

Often, we need to perform an operation over every element in a variadic template parameter pack. There are many ways to do this, and the solutions get easier to read and write with C++17. Suppose we simply want to print every element in a pack. The simplest solution is to recurse:

Version ≥ C++11

```
void print_all(std::ostream& os) {
    // base case
}

template <class T, class... Ts>
void print_all(std::ostream& os, T const& first, Ts const&... rest) {
    os << first;

    print_all(os, rest...);
}
```

We could instead use the expander trick, to perform all the streaming in a single function. This has the advantage of not needing a second overload, but has the disadvantage of less than stellar readability:

Version ≥ C++11

```
template <class... Ts>
void print_all(std::ostream& os, Ts const&... args) {
    using expander = int[];
    (void)expander{0,
        (void(os << args), 0)...
    };
}
```

For an explanation of how this works, see T.C's excellent answer.

Version ≥ C++17

With C++17, we get two powerful new tools in our arsenal for solving this problem. The first is a fold-expression:

```
template <class... Ts>
void print_all(std::ostream& os, Ts const&... args) {
    ((os << args), ...);
}
```

And the second is `if constexpr`, which allows us to write our original recursive solution in a single function:

```
template <class T, class... Ts>
void print_all(std::ostream& os, T const& first, Ts const&... rest) {
    os << first;

    if constexpr (sizeof...(rest) > 0) {
        // this line will only be instantiated if there are further
        // arguments. if rest... is empty, there will be no call to
        // print_all(os).
        print_all(os, rest...);
    }
}
```

Section 16.3: Iterating with std::integer_sequence

Since C++14, the standard provides the class template

```
template <class T, T... Ints>
class integer_sequence;

template <std::size_t... Ints>
using index_sequence = std::integer_sequence<std::size_t, Ints...>;
```

and a generating metafunction for it:

```
template <class T, T N>
using make_integer_sequence = std::integer_sequence<T, /* a sequence 0, 1, 2, ..., N-1 */ >;

template<std::size_t N>
using make_index_sequence = make_integer_sequence<std::size_t, N>;
```

While this comes standard in C++14, this can be implemented using C++11 tools.

We can use this tool to call a function with a `std::tuple` of arguments (standardized in C++17 as `std::apply`):

```
namespace detail {
    template <class F, class Tuple, std::size_t... Is>
    decltype(auto) apply_impl(F&& f, Tuple&& tpl, std::index_sequence<Is...> ) {
        return std::forward<F>(f)(std::get<Is>(std::forward<Tuple>(tpl))...);
    }
}

template <class F, class Tuple>
decltype(auto) apply(F&& f, Tuple&& tpl) {
    return detail::apply_impl(std::forward<F>(f),
        std::forward<Tuple>(tpl),
        std::make_index_sequence<std::tuple_size<std::decay_t<Tuple>>::value>{});
}


// this will print 3
int f(int, char, double);
```

```
auto some_args = std::make_tuple(42, 'x', 3.14);
int r = apply(f, some_args); // calls f(42, 'x', 3.14)
```

Section 16.4: Tag Dispatching

A simple way of selecting between functions at compile time is to dispatch a function to an overloaded pair of functions that take a tag as one (usually the last) argument. For example, to implement `std::advance()`, we can dispatch on the iterator category:

```
namespace details {
    template <class RAIter, class Distance>
    void advance(RAIter& it, Distance n, std::random_access_iterator_tag) {
        it += n;
    }

    template <class BidirIter, class Distance>
    void advance(BidirIter& it, Distance n, std::bidirectional_iterator_tag) {
        if (n > 0) {
            while (n--) ++it;
        }
        else {
            while (n++) --it;
        }
    }

    template <class InputIter, class Distance>
    void advance(InputIter& it, Distance n, std::input_iterator_tag) {
        while (n--) {
            ++it;
        }
    }
}

template <class Iter, class Distance>
void advance(Iter& it, Distance n) {
    details::advance(it, n,
            typename std::iterator_traits<Iter>::iterator_category{} );
}
```

The `std::XY_iterator_tag` arguments of the overloaded `details::advance` functions are unused function parameters. The actual implementation does not matter (actually it is completely empty). Their only purpose is to allow the compiler to select an overload based on which tag class `details::advance` is called with.

In this example, `advance` uses the `iterator_traits<T>::iterator_category` metafunction which returns one of the `iterator_tag` classes, depending on the actual type of `Iter`. A default-constructed object of the `iterator_category<Iter>::type` then lets the compiler select one of the different overloads of `details::advance`. (This function parameter is likely to be completely optimized away, as it is a default-constructed object of an empty `struct` and never used.)

Tag dispatching can give you code that's much easier to read than the equivalents using SFINAE and `enable_if`.

Note: while C++17's `if constexpr` may simplify the implementation of `advance` in particular, it is not suitable for open implementations unlike tag dispatching.

Section 16.5: Detect Whether Expression is Valid

It is possible to detect whether an operator or function can be called on a type. To test if a class has an overload of

std::hash, one can do this:

```
#include <functional> // for std::hash
#include <type_traits> // for std::false_type and std::true_type
#include <utility> // for std::declval

template<class, class = void>
struct has_hash
    : std::false_type
{};

template<class T>
struct has_hash<T, decltype(std::hash<T>()(std::declval<T>()), void())>
    : std::true_type
{};
```

Version ≥ C++17

Since C++17, std::void_t can be used to simplify this type of construct

```
#include <functional> // for std::hash
#include <type_traits> // for std::false_type, std::true_type, std::void_t
#include <utility> // for std::declval

template<class, class = std::void_t<> >
struct has_hash
    : std::false_type
{};

template<class T>
struct has_hash<T, std::void_t< decltype(std::hash<T>()(std::declval<T>())) > >
    : std::true_type
{};
```

where std::void_t is defined as:

```
template< class... > using void_t = void;
```

For detecting if an operator, such as operator< is defined, the syntax is almost the same:

```
template<class, class = void>
struct has_less_than
    : std::false_type
{};

template<class T>
struct has_less_than<T, decltype(std::declval<T>() < std::declval<T>(), void())>
    : std::true_type
{};
```

These can be used to use a std::unordered_map<T> if T has an overload for std::hash, but otherwise attempt to use a std::map<T>:

```
template <class K, class V>
using hash_invariant_map = std::conditional_t<
    has_hash<K>::value,
    std::unordered_map<K, V>,
    std::map<K,V>>;
```

Section 16.6: If-then-else

Version ≥ C++11

The type `std::conditional` in the standard library header **<type_traits>** can select one type or the other, based on a compile-time boolean value:

```
template<typename T>
struct ValueOrPointer
{
    typename std::conditional<(sizeof(T) > sizeof(void*)), T*, T>::type vop;
};
```

This struct contains a pointer to `T` if `T` is larger than the size of a pointer, or `T` itself if it is smaller or equal to a pointer's size. Therefore `sizeof(ValueOrPointer)` will always be <= `sizeof(void*)`.

Section 16.7: Manual distinction of types when given any type T

When implementing SFINAE using `std::enable_if`, it is often useful to have access to helper templates that determines if a given type `T` matches a set of criteria.

To help us with that, the standard already provides two types analog to `true` and `false` which are `std::true_type` and `std::false_type`.

The following example show how to detect if a type `T` is a pointer or not, the `is_pointer` template mimic the behavior of the standard `std::is_pointer` helper:

```
template <typename T>
struct is_pointer_: std::false_type {};

template <typename T>
struct is_pointer_<T*>: std::true_type {};

template <typename T>
struct is_pointer: is_pointer_<typename std::remove_cv<T>::type> { }
```

There are three steps in the above code (sometimes you only need two):

1. The first declaration of `is_pointer_` is the *default case*, and inherits from `std::false_type`. The *default* case should always inherit from `std::false_type` since it is analogous to a "`false` condition".
2. The second declaration specialize the `is_pointer_` template for pointer `T*` without caring about what `T` is really. This version inherits from `std::true_type`.
3. The third declaration (the real one) simply remove any unnecessary information from `T` (in this case we remove `const` and `volatile` qualifiers) and then fall backs to one of the two previous declarations.

Since `is_pointer<T>` is a class, to access its value you need to either:

- Use `::value`, e.g. `is_pointer<int>::value` – `value` is a static class member of type `bool` inherited from `std::true_type` or `std::false_type`;
- Construct an object of this type, e.g. `is_pointer<int>{}` – This works because `std::is_pointer` inherits its default constructor from `std::true_type` or `std::false_type` (which have `constexpr` constructors) and both `std::true_type` and `std::false_type` have `constexpr` conversion operators to `bool`.

It is a good habit to provides "helper helper templates" that let you directly access the value:

```
template <typename T>
constexpr bool is_pointer_v = is_pointer<T>::value;
```

Version ≥ C++17

In C++17 and above, most helper templates already provide a `_v` version, e.g.:

```
template< class T > constexpr bool is_pointer_v = is_pointer<T>::value;
template< class T > constexpr bool is_reference_v = is_reference<T>::value;
```

Section 16.8: Calculating power with C++11 (and higher)

With C++11 and higher calculations at compile time can be much easier. For example calculating the power of a given number at compile time will be following:

```
template <typename T>
constexpr T calculatePower(T value, unsigned power) {
    return power == 0 ? 1 : value * calculatePower(value, power-1);
}
```

Keyword `constexpr` is responsible for calculating function in compilation time, then and only then, when all the requirements for this will be met (see more at constexpr keyword reference) for example all the arguments must be known at compile time.

Note: In C++11 `constexpr` function must compose only from one return statement.

Advantages: Comparing this to the standard way of compile time calculation, this method is also useful for runtime calculations. It means, that if the arguments of the function are not known at the compilation time (e.g. value and power are given as input via user), then function is run in a compilation time, so there's no need to duplicate a code (as we would be forced in older standards of C++).

E.g.

```
void useExample() {
    constexpr int compileTimeCalculated = calculatePower(3, 3); // computes at compile time,
                                 // as both arguments are known at compilation time
                                 // and used for a constant expression.
    int value;
    std::cin >> value;
    int runtimeCalculated = calculatePower(value, 3);  // runtime calculated,
                                        // because value is known only at runtime.
}
```

Version ≥ C++17

Another way to calculate power at compile time can make use of fold expression as follows:

```
#include <iostream>
#include <utility>

template <class T, T V, T N, class I = std::make_integer_sequence<T, N>>
struct power;

template <class T, T V, T N, T... Is>
struct power<T, V, N, std::integer_sequence<T, Is...>> {
   static constexpr T value = (static_cast<T>(1) * ... * (V * static_cast<bool>(Is + 1)));
};
```

```
int main() {
   std::cout << power<int, 4, 2>::value << std::endl;
}
```

Section 16.9: Generic Min/Max with variable argument count

Version > C++11

It's possible to write a generic function (for example `min`) which accepts various numerical types and arbitrary argument count by template meta-programming. This function declares a `min` for two arguments and recursively for more.

```
template <typename T1, typename T2>
auto min(const T1 &a, const T2 &b)
-> typename std::common_type<const T1&, const T2&>::type
{
    return a < b ? a : b;
}

template <typename T1, typename T2, typename ... Args>
auto min(const T1 &a, const T2 &b, const Args& ... args)
-> typename std::common_type<const T1&, const T2&, const Args& ...>::type
{
    return min(min(a, b), args...);
}

auto minimum = min(4, 5.8f, 3, 1.8, 3, 1.1, 9);
```

Chapter 17: const keyword

Section 17.1: Avoiding duplication of code in const and non-const getter methods

In C++ methods that differs only by `const` qualifier can be overloaded. Sometimes there may be a need of two versions of getter that return a reference to some member.

Let `Foo` be a class, that has two methods that perform identical operations and returns a reference to an object of type `Bar`:

```
class Foo
{
public:
    Bar& GetBar(/* some arguments */)
    {
        /* some calculations */
        return bar;
    }

    const Bar& GetBar(/* some arguments */) const
    {
        /* some calculations */
        return bar;
    }

    // ...
};
```

The only difference between them is that one method is non-const and return a non-const reference (that can be use to modify object) and the second is const and returns const reference.

To avoid the code duplication, there is a temptation to call one method from another. However, we can not call non-const method from the const one. But we can call const method from non-const one. That will require as to use 'const_cast' to remove the const qualifier.

The solution is:

```
struct Foo
{
    Bar& GetBar(/*arguments*/)
    {
        return const_cast<Bar&>(const_cast<const Foo*>(this)->GetBar(/*arguments*/));
    }

    const Bar& GetBar(/*arguments*/) const
    {
        /* some calculations */
        return foo;
    }
};
```

In code above, we call const version of `GetBar` from the non-const `GetBar` by casting this to const type: `const_cast<const Foo*>(this)`. Since we call const method from non-const, the object itself is non-const, and casting away the const is allowed.

Examine the following more complete example:

```
#include <iostream>

class Student
{
public:
    char& GetScore(bool midterm)
    {
        return const_cast<char&>(const_cast<const Student*>(this)->GetScore(midterm));
    }

    const char& GetScore(bool midterm) const
    {
        if (midterm)
        {
            return midtermScore;
        }
        else
        {
            return finalScore;
        }
    }

private:
    char midtermScore;
    char finalScore;
};

int main()
{
    // non-const object
    Student a;
    // We can assign to the reference. Non-const version of GetScore is called
    a.GetScore(true) = 'B';
    a.GetScore(false) = 'A';

    // const object
    const Student b(a);
    // We still can call GetScore method of const object,
    // because we have overloaded const version of GetScore
    std::cout << b.GetScore(true) << b.GetScore(false) << '\n';
}
```

Section 17.2: Const member functions

Member functions of a class can be declared `const`, which tells the compiler and future readers that this function will not modify the object:

```
class MyClass
{
private:
    int myInt_;
public:
    int myInt() const { return myInt_; }
    void setMyInt(int myInt) { myInt_ = myInt; }
};
```

In a `const` member function, the `this` pointer is effectively a `const MyClass *` instead of a `MyClass *`. This means that you cannot change any member variables within the function; the compiler will emit a warning. So `setMyInt`

could not be declared `const`.

You should almost always mark member functions as `const` when possible. Only `const` member functions can be called on a `const MyClass`.

`static` methods cannot be declared as `const`. This is because a static method belongs to a class and is not called on object; therefore it can never modify object's internal variables. So declaring `static` methods as `const` would be redundant.

Section 17.3: Const local variables

Declaration and usage.

```
// a is const int, so it can't be changed
const int a = 15;
a = 12;           // Error: can't assign new value to const variable
a += 1;           // Error: can't assign new value to const variable
```

Binding of references and pointers

```
int &b = a;       // Error: can't bind non-const reference to const variable
const int &c = a; // OK; c is a const reference

int *d = &a;      // Error: can't bind pointer-to-non-const to const variable
const int *e = &a // OK; e is a pointer-to-const

int f = 0;
e = &f;           // OK; e is a non-const pointer-to-const,
                  // which means that it can be rebound to new int* or const int*

*e = 1            // Error: e is a pointer-to-const which means that
                  // the value it points to can't be changed through dereferencing e

int *g = &f;
*g = 1;           // OK; this value still can be changed through dereferencing
                  // a pointer-not-to-const
```

Section 17.4: Const pointers

```
int a = 0, b = 2;

const int* pA = &a; // pointer-to-const. `a` can't be changed through this
int* const pB = &a; // const pointer. `a` can be changed, but this pointer can't.
const int* const pC = &a; // const pointer-to-const.

//Error: Cannot assign to a const reference
*pA = b;

pA = &b;

*pB = b;

//Error: Cannot assign to const pointer
pB = &b;

//Error: Cannot assign to a const reference
*pC = b;
```

```
//Error: Cannot assign to const pointer
pC = &b;
```

Chapter 18: mutable keyword

Section 18.1: mutable lambdas

By default, the implicit `operator()` of a lambda is `const`. This disallows performing non-`const` operations on the lambda. In order to allow modifying members, a lambda may be marked `mutable`, which makes the implicit `operator()` non-`const`:

```
int a = 0;

auto bad_counter = [a] {
    return a++;   // error: operator() is const
                  // cannot modify members
};

auto good_counter = [a]() mutable {
    return a++;  // OK
}

good_counter(); // 0
good_counter(); // 1
good_counter(); // 2
```

Section 18.2: non-static class member modifier

`mutable` modifier in this context is used to indicate that a data field of a const object may be modified without affecting the externally-visible state of the object.

If you are thinking about caching a result of expensive computation, you should probably use this keyword.

If you have a lock (for example, `std::unique_lock`) data field which is locked and unlocked inside a const method, this keyword is also what you could use.

You should not use this keyword to break logical const-ness of an object.

Example with caching:

```
class pi_calculator {
 public:
     double get_pi() const {
         if (pi_calculated) {
             return pi;
         } else {
             double new_pi = 0;
             for (int i = 0; i < 1000000000; ++i) {
                 // some calculation to refine new_pi
             }
             // note: if pi and pi_calculated were not mutable, we would get an error from a
compiler
             // because in a const method we can not change a non-mutable field
             pi = new_pi;
             pi_calculated = true;
             return pi;
         }
     }
 private:
     mutable bool pi_calculated = false;
```

```
    mutable double pi = 0;
};
```

Chapter 19: Friend keyword

Well-designed classes encapsulate their functionality, hiding their implementation while providing a clean, documented interface. This allows redesign or change so long as the interface is unchanged.

In a more complex scenario, multiple classes that rely on each others' implementation details may be required. Friend classes and functions allow these peers access to each others' details, without compromising the encapsulation and information hiding of the documented interface.

Section 19.1: Friend function

A class or a structure may declare any function it's friend. If a function is a friend of a class, it may access all it's protected and private members:

```
// Forward declaration of functions.
void friend_function();
void non_friend_function();

class PrivateHolder {
public:
    PrivateHolder(int val) : private_value(val) {}
private:
    int private_value;
    // Declare one of the function as a friend.
    friend void friend_function();
};

void non_friend_function() {
    PrivateHolder ph(10);
    // Compilation error: private_value is private.
    std::cout << ph.private_value << std::endl;
}

void friend_function() {
    // OK: friends may access private values.
    PrivateHolder ph(10);
    std::cout << ph.private_value << std::endl;
}
```

Access modifiers do not alter friend semantics. Public, protected and private declarations of a friend are equivalent.

Friend declarations are not inherited. For example, if we subclass `PrivateHolder`:

```
class PrivateHolderDerived : public PrivateHolder {
public:
    PrivateHolderDerived(int val) : PrivateHolder(val) {}
private:
    int derived_private_value = 0;
};
```

and try to access it's members, we'll get the following:

```
void friend_function() {
    PrivateHolderDerived pd(20);
    // OK.
    std::cout << pd.private_value << std::endl;
    // Compilation error: derived_private_value is private.
```

```
    std::cout << pd.derived_private_value << std::endl;
}
```

Note that `PrivateHolderDerived` member function cannot access `PrivateHolder::private_value`, while friend function can do it.

Section 19.2: Friend method

Methods may declared as friends as well as functions:

```
class Accesser {
public:
    void private_accesser();
};

class PrivateHolder {
public:
    PrivateHolder(int val) : private_value(val) {}
    friend void Accesser::private_accesser();
private:
    int private_value;
};

void Accesser::private_accesser() {
    PrivateHolder ph(10);
    // OK: this method is declares as friend.
    std::cout << ph.private_value << std::endl;
}
```

Section 19.3: Friend class

A whole class may be declared as friend. Friend class declaration means that any member of the friend may access private and protected members of the declaring class:

```
class Accesser {
public:
    void private_accesser1();
    void private_accesser2();
};

class PrivateHolder {
public:
    PrivateHolder(int val) : private_value(val) {}
    friend class Accesser;
private:
    int private_value;
};

void Accesser::private_accesser1() {
    PrivateHolder ph(10);
    // OK.
    std::cout << ph.private_value << std::endl;
}

void Accesser::private_accesser2() {
    PrivateHolder ph(10);
    // OK.
    std::cout << ph.private_value + 1 << std::endl;
```

```
}
```

Friend class declaration is not reflexive. If classes need private access in both directions, both of them need friend declarations.

```
class Accesser {
public:
    void private_accesser1();
    void private_accesser2();
private:
    int private_value = 0;
};

class PrivateHolder {
public:
    PrivateHolder(int val) : private_value(val) {}
    // Accesser is a friend of PrivateHolder
    friend class Accesser;
    void reverse_accesse() {
        // but PrivateHolder cannot access Accesser's members.
        Accesser a;
        std::cout << a.private_value;
    }
private:
    int private_value;
};
```

Chapter 20: Type Keywords

Section 20.1: class

1. Introduces the definition of a class type.

```
class foo {
    int x;
  public:
    int get_x();
    void set_x(int new_x);
};
```

2. Introduces an *elaborated type specifier,* which specifies that the following name is the name of a class type. If the class name has been declared already, it can be found even if hidden by another name. If the class name has not been declared already, it is forward-declared.

```
class foo; // elaborated type specifier -> forward declaration
class bar {
  public:
    bar(foo& f);
};
void baz();
class baz; // another elaborated type specifer; another forward declaration
           // note: the class has the same name as the function void baz()
class foo {
    bar b;
    friend class baz; // elaborated type specifier refers to the class,
                      // not the function of the same name
  public:
    foo();
};
```

3. Introduces a type parameter in the declaration of a template.

```
template <class T>
const T& min(const T& x, const T& y) {
    return b < a ? b : a;
}
```

4. In the declaration of a template template parameter, the keyword `class` precedes the name of the parameter. Since the argument for a template template parameter can only be a class template, the use of `class` here is redundant. However, the grammar of C++ requires it.

```
template <template <class T> class U>
//                           ^^^^^ "class" used in this sense here;
//                                 U is a template template parameter
void f() {
    U<int>::do_it();
    U<double>::do_it();
}
```

5. Note that sense 2 and sense 3 may be combined in the same declaration. For example:

```
template <class T>
class foo {
```

```
};

foo<class bar> x; // <- bar does not have to have previously appeared.
```

Version ≥ C++11

6. In the declaration or definition of an enum, declares the enum to be a scoped enum.

```
enum class Format {
    TEXT,
    PDF,
    OTHER,
};
Format f = F::TEXT;
```

Section 20.2: enum

1. Introduces the definition of an enumeration type.

```
enum Direction {
    UP,
    LEFT,
    DOWN,
    RIGHT
};
Direction d = UP;
```

Version ≥ C++11

In C++11, `enum` may optionally be followed by `class` or `struct` to define a scoped enum. Furthermore, both scoped and unscoped enums can have their underlying type explicitly specified by `: T` following the enum name, where `T` refers to an integer type.

```
enum class Format : char {
    TEXT,
    PDF,
    OTHER
};
Format f = Format::TEXT;

enum Language : int {
    ENGLISH,
    FRENCH,
    OTHER
};
```

Enumerators in normal `enum`s may also be preceded by the scope operator, although they are still considered to be in the scope the `enum` was defined in.

```
Language l1, l2;

l1 = ENGLISH;
l2 = Language::OTHER;
```

2. Introduces an *elaborated type specifier*, which specifies that the following name is the name of a previously declared enum type. (An elaborated type specifier cannot be used to forward-declare an enum type.) An

enum can be named in this way even if hidden by another name.

```
enum Foo { FOO };
void Foo() {}
Foo foo = FOO;      // ill-formed; Foo refers to the function
enum Foo foo = FOO; // ok; Foo refers to the enum type
```

Version ≥ C++11

3. Introduces an *opaque enum declaration,* which declares an enum without defining it. It can either redeclare a previously declared enum, or forward-declare an enum that has not been previously declared.

   An enum first declared as scoped cannot later be declared as unscoped, or *vice versa.* All declarations of an enum must agree in underlying type.

   When forward-declaring an unscoped enum, the underlying type must be explicitly specified, since it cannot be inferred until the values of the enumerators are known.

```
enum class Format; // underlying type is implicitly int
void f(Format f);
enum class Format {
    TEXT,
    PDF,
    OTHER,
};

enum Direction;    // ill-formed; must specify underlying type
```

Section 20.3: struct

Interchangeable with `class`, except for the following differences:

- If a class type is defined using the keyword `struct`, then the default accessibility of bases and members is `public` rather than `private`.
- `struct` cannot be used to declare a template type parameter or template template parameter; only `class` can.

Section 20.4: union

1. Introduces the definition of a union type.

```
// Example is from POSIX
union sigval {
    int     sival_int;
    void   *sival_ptr;
};
```

2. Introduces an *elaborated type specifier,* which specifies that the following name is the name of a union type. If the union name has been declared already, it can be found even if hidden by another name. If the union name has not been declared already, it is forward-declared.

```
union foo; // elaborated type specifier -> forward declaration
class bar {
```

```
  public:
    bar(foo& f);
};
void baz();
union baz; // another elaborated type specifer; another forward declaration
           // note: the class has the same name as the function void baz()
union foo {
    long l;
    union baz* b; // elaborated type specifier refers to the class,
                  // not the function of the same name
};
```

Chapter 21: Basic Type Keywords

Section 21.1: char

An integer type which is "large enough to store any member of the implementation's basic character set". It is implementation-defined whether `char` is signed (and has a range of at least -127 to +127, inclusive) or unsigned (and has a range of at least 0 to 255, inclusive).

```
const char zero = '0';
const char one = zero + 1;
const char newline = '\n';
std::cout << one << newline; // prints 1 followed by a newline
```

Section 21.2: char16_t

Version ≥ C++11

An unsigned integer type with the same size and alignment as `uint_least16_t`, which is therefore large enough to hold a UTF-16 code unit.

```
const char16_t message[] = u"你好，世界\\n";            // Chinese for "hello, world\\n"
std::cout << sizeof(message)/sizeof(char16_t) << "\\n"; // prints 7
```

Section 21.3: char32_t

Version ≥ C++11

An unsigned integer type with the same size and alignment as `uint_least32_t`, which is therefore large enough to hold a UTF-32 code unit.

```
const char32_t full_house[] = U"□□□□□";               // non-BMP characters
std::cout << sizeof(full_house)/sizeof(char32_t) << "\\n"; // prints 6
```

Section 21.4: int

Denotes a signed integer type with "the natural size suggested by the architecture of the execution environment", whose range includes at least -32767 to +32767, inclusive.

```
int x = 2;
int y = 3;
int z = x + y;
```

Can be combined with `unsigned`, `short`, `long`, and `long long` (q.v.) in order to yield other integer types.

Section 21.5: void

An incomplete type; it is not possible for an object to have type `void`, nor are there arrays of `void` or references to `void`. It is used as the return type of functions that do not return anything.

Moreover, a function may redundantly be declared with a single parameter of type `void`; this is equivalent to declaring a function with no parameters (e.g. `int main()` and `int main(void)` declare the same function). This syntax is allowed for compatibility with C (where function declarations have a different meaning than in C++).

The type `void*` ("pointer to `void`") has the property that any object pointer can be converted to it and back and result in the same pointer. This feature makes the type `void*` suitable for certain kinds of (type-unsafe) type-erasing interfaces, for example for generic contexts in C-style APIs (e.g. `qsort`, `pthread_create`).

Any expression may be converted to an expression of type `void`; this is called a *discarded-value expression*:

```
static_cast<void>(std::printf("Hello, %s!\n", name));  // discard return value
```

This may be useful to signal explicitly that the value of an expression is not of interest and that the expression is to be evaluated for its side effects only.

Section 21.6: wchar_t

An integer type large enough to represent all characters of the largest supported extended character set, also known as the wide-character set. (It is not portable to make the assumption that `wchar_t` uses any particular encoding, such as UTF-16.)

It is normally used when you need to store characters over ASCII 255 , as it has a greater size than the character type `char`.

```
const wchar_t message_ahmaric[] = L"□□□ □□□\\n"; //Ahmaric for "hello, world\\n"
const wchar_t message_chinese[] = L"你好，世界\\n";// Chinese for "hello, world\\n"
const wchar_t message_hebrew[]  = L"שלום עולם\\n"; //Hebrew for "hello, world\\n"
const wchar_t message_russian[] = L"Привет мир\\n";  //Russian for "hello, world\\n"
const wchar_t message_tamil[]   = L"ஹலோ உலகம்\\n"; //Tamil for "hello, world\\n"
```

Section 21.7: float

A floating point type. Has the narrowest range out of the three floating point types in C++.

```
float area(float radius) {
    const float pi = 3.14159f;
    return pi*radius*radius;
}
```

Section 21.8: double

A floating point type. Its range includes that of `float`. When combined with `long`, denotes the `long double` floating point type, whose range includes that of `double`.

```
double area(double radius) {
    const double pi = 3.141592653589793;
    return pi*radius*radius;
}
```

Section 21.9: long

Denotes a signed integer type that is at least as long as `int`, and whose range includes at least -2147483647 to +2147483647, inclusive (that is, -(2^31 - 1) to +(2^31 - 1)). This type can also be written as `long int`.

```
const long approx_seconds_per_year = 60L*60L*24L*365L;
```

The combination `long double` denotes a floating point type, which has the widest range out of the three floating

point types.

```
long double area(long double radius) {
    const long double pi = 3.1415926535897932385L;
    return pi*radius*radius;
}
```

Version ≥ C++11

When the `long` specifier occurs twice, as in `long long`, it denotes a signed integer type that is at least as long as `long`, and whose range includes at least -9223372036854775807 to +9223372036854775807, inclusive (that is, -(2^63 - 1) to +(2^63 - 1)).

```
// support files up to 2 TiB
const long long max_file_size = 2LL << 40;
```

Section 21.10: short

Denotes a signed integer type that is at least as long as `char`, and whose range includes at least -32767 to +32767, inclusive. This type can also be written as `short int`.

```
// (during the last year)
short hours_worked(short days_worked) {
    return 8*days_worked;
}
```

Section 21.11: bool

An integer type whose value can be either `true` or `false`.

```
bool is_even(int x) {
    return x%2 == 0;
}
const bool b = is_even(47); // false
```

Chapter 22: Variable Declaration Keywords

Section 22.1: decltype

Version ≥ C++11

Yields the type of its operand, which is not evaluated.

- If the operand `e` is a name without any additional parentheses, then `decltype(e)` is the *declared type* of `e`.

```
int x = 42;
std::vector<decltype(x)> v(100, x); // v is a vector<int>
```

- If the operand `e` is a class member access without any additional parentheses, then `decltype(e)` is the *declared type* of the member accessed.

```
struct S {
    int x = 42;
};
const S s;
decltype(s.x) y; // y has type int, even though s.x is const
```

- In all other cases, `decltype(e)` yields both the type and the value category of the expression `e`, as follows:
  - If `e` is an lvalue of type `T`, then `decltype(e)` is `T&`.
  - If `e` is an xvalue of type `T`, then `decltype(e)` is `T&&`.
  - If `e` is a prvalue of type `T`, then `decltype(e)` is `T`.

  This includes the case with extraneous parentheses.

```
int f() { return 42; }
int& g() { static int x = 42; return x; }
int x = 42;
decltype(f()) a = f(); // a has type int
decltype(g()) b = g(); // b has type int&
decltype((x)) c = x;   // c has type int&, since x is an lvalue
```

Version ≥ C++14

The special form `decltype(auto)` deduces the type of a variable from its initializer or the return type of a function from the `return` statements in its definition, using the type deduction rules of `decltype` rather than those of `auto`.

```
const int x = 123;
auto y = x;           // y has type int
decltype(auto) z = x; // z has type const int, the declared type of x
```

Section 22.2: const

A type specifier; when applied to a type, produces the const-qualified version of the type. See const keyword for details on the meaning of `const`.

```
const int x = 123;
x = 456;    // error
```

```
int& r = x; // error

struct S {
    void f();
    void g() const;
};
const S s;
s.f(); // error
s.g(); // OK
```

Section 22.3: volatile

A type qualifier; when applied to a type, produces the volatile-qualified version of the type. Volatile qualification plays the same role as `const` qualification in the type system, but `volatile` does not prevent objects from being modified; instead, it forces the compiler to treat all accesses to such objects as side effects.

In the example below, if `memory_mapped_port` were not volatile, the compiler could optimize the function so that it performs only the final write, which would be incorrect if `sizeof(int)` is greater than 1. The `volatile` qualification forces it to treat all `sizeof(int)` writes as different side effects and hence perform all of them (in order).

```
extern volatile char memory_mapped_port;
void write_to_device(int x) {
    const char* p = reinterpret_cast<const char*>(&x);
    for (int i = 0; i < sizeof(int); i++) {
        memory_mapped_port = p[i];
    }
}
```

Section 22.4: signed

A keyword that is part of certain integer type names.

- When used alone, `int` is implied, so that `signed`, `signed int`, and `int` are the same type.
- When combined with `char`, yields the type `signed char`, which is a different type from `char`, even if `char` is also signed. `signed char` has a range that includes at least -127 to +127, inclusive.
- When combined with `short`, `long`, or `long long`, it is redundant, since those types are already signed.
- `signed` cannot be combined with `bool`, `wchar_t`, `char16_t`, or `char32_t`.

Example:

```
signed char celsius_temperature;
std::cin >> celsius_temperature;
if (celsius_temperature < -35) {
    std::cout << "cold day, eh?\n";
}
```

Section 22.5: unsigned

A type specifier that requests the unsigned version of an integer type.

- When used alone, `int` is implied, so `unsigned` is the same type as `unsigned int`.
- The type `unsigned char` is different from the type `char`, even if `char` is unsigned. It can hold integers up to at least 255.
- `unsigned` can also be combined with `short`, `long`, or `long long`. It cannot be combined with `bool`, `wchar_t`, `char16_t`, or `char32_t`.

Example:

```
char invert_case_table[256] = { ..., 'a', 'b', 'c', ..., 'A', 'B', 'C', ... };
char invert_case(char c) {
    unsigned char index = c;
    return invert_case_table[index];
    // note: returning invert_case_table[c] directly does the
    // wrong thing on implementations where char is a signed type
}
```

Chapter 23: Keywords

Keywords have fixed meaning defined by the C++ standard and cannot be used as identifiers. It is illegal to redefine keywords using the preprocessor in any translation unit that includes a standard library header. However, keywords lose their special meaning inside attributes.

Section 23.1: asm

The `asm` keyword takes a single operand, which must be a string literal. It has an implementation-defined meaning, but is typically passed to the implementation's assembler, with the assembler's output being incorporated into the translation unit.

The `asm` statement is a *definition*, not an *expression*, so it may appear either at block scope or namespace scope (including global scope). However, since inline assembly cannot be constrained by the rules of the C++ language, `asm` may not appear inside a `constexpr` function.

Example:

```
[[noreturn]] void halt_system() {
    asm("hlt");
}
```

Section 23.2: Different keywords

> **void C++**

1. When used as a function return type, the void keyword specifies that the function does not return a value. When used for a function's parameter list, void specifies that the function takes no parameters. When used in the declaration of a pointer, void specifies that the pointer is "universal."
2. If a pointer's type is void *, the pointer can point to any variable that is not declared with the const or volatile keyword. A void pointer cannot be dereferenced unless it is cast to another type. A void pointer can be converted into any other type of data pointer.
3. A void pointer can point to a function, but not to a class member in C++.

```
void vobject;   // C2182
void *pv;   // okay
int *pint; int i;
int main() {
pv = &i;
   // Cast optional in C required in C++
pint = (int *)pv;
```

> **Volatile C++**

1. A type qualifier that you can use to declare that an object can be modified in the program by the hardware.

```
volatile declarator ;
```

virtual C++

1. The virtual keyword declares a virtual function or a virtual base class.

```
virtual [type-specifiers] member-function-declarator
virtual [access-specifier] base-class-name
```

Parameters

1. **type-specifiers** Specifies the return type of the virtual member function.
2. **member-function-declarator** Declares a member function.
3. **access-specifier** Defines the level of access to the base class, public, protected or private. Can appear before or after the virtual keyword.
4. **base-class-name** Identifies a previously declared class type

this pointer

1. The this pointer is a pointer accessible only within the nonstatic member functions of a class, struct, or union type. It points to the object for which the member function is called. Static member functions do not have a this pointer.

```
this->member-identifier
```

An object's this pointer is not part of the object itself; it is not reflected in the result of a sizeof statement on the object. Instead, when a nonstatic member function is called for an object, the address of the object is passed by the compiler as a hidden argument to the function. For example, the following function call:

```
myDate.setMonth( 3 );
```

can be interpreted this way:

```
setMonth( &myDate, 3 );
```

The object's address is available from within the member function as the this pointer. Most uses of this are implicit. It is legal, though unnecessary, to explicitly use this when referring to members of the class. For example:

```
void Date::setMonth( int mn )
{
   month = mn;            // These three statements
   this->month = mn;      // are equivalent
   (*this).month = mn;
}
```

The expression *this is commonly used to return the current object from a member function: return *this; The this pointer is also used to guard against self-reference:

```
if (&Object != this) {
// do not execute in cases of self-reference
```

try, throw, and catch Statements (C++)

1. To implement exception handling in C++, you use try, throw, and catch expressions.
2. First, use a try block to enclose one or more statements that might throw an exception.
3. A throw expression signals that an exceptional condition—often, an error—has occurred in a try block. You can use an object of any type as the operand of a throw expression. Typically, this object is used to communicate information about the error. In most cases, we recommend that you use the std::exception class or one of the derived classes that are defined in the standard library. If one of those is not appropriate, we recommend that you derive your own exception class from std::exception.
4. To handle exceptions that may be thrown, implement one or more catch blocks immediately following a try block. Each catch block specifies the type of exception it can handle.

```
    MyData md;
try {
   // Code that could throw an exception
   md = GetNetworkResource();
}
catch (const networkIOException& e) {
   // Code that executes when an exception of type
   // networkIOException is thrown in the try block
   // ...
   // Log error message in the exception object
   cerr << e.what();
}
catch (const myDataFormatException& e) {
   // Code that handles another exception type
   // ...
   cerr << e.what();
}

// The following syntax shows a throw expression
MyData GetNetworkResource()
{
   // ...
   if (IOSuccess == false)
      throw networkIOException("Unable to connect");
   // ...
   if (readError)
      throw myDataFormatException("Format error");
   // ...
}
```

> The code after the try clause is the guarded section of code. The throw expression throws—that is, raises—an exception. The code block after the catch clause is the exception handler. This is the handler that catches the exception that's thrown if the types in the throw and catch expressions are compatible.

```
    try {
   throw CSomeOtherException();
}
catch(...) {
   // Catch all exceptions – dangerous!!!
   // Respond (perhaps only partially) to the exception, then
   // re-throw to pass the exception to some other handler
   // ...
   throw;
}
```

> **friend (C++)**

1. In some circumstances, it is more convenient to grant member-level access to functions that are not members of a class or to all members in a separate class. Only the class implementer can declare who its friends are. A function or class cannot declare itself as a friend of any class. In a class definition, use the friend keyword and the name of a non-member function or other class to grant it access to the private and protected members of your class. In a template definition, a type parameter can be declared as a friend.

2. If you declare a friend function that was not previously declared, that function is exported to the enclosing nonclass scope.

```
class friend F
friend F;
class ForwardDeclared;// Class name is known.
class HasFriends
{
   friend int ForwardDeclared::IsAFriend();// C2039 error expected
};
```

friend functions

1. A friend function is a function that is not a member of a class but has access to the class's private and protected members.Friend functions are not considered class members; they are normal external functions that are given special access privileges.

2. Friends are not in the class's scope, and they are not called using the member-selection operators (. and –>) unless they are members of another class.

3. A friend function is declared by the class that is granting access. The friend declaration can be placed anywhere in the class declaration. It is not affected by the access control keywords.

```
#include <iostream>

using namespace std;
class Point
{
    friend void ChangePrivate( Point & );
public:
    Point( void ) : m_i(0) {}
    void PrintPrivate( void ){cout << m_i << endl; }

private:
int m_i;
};

void ChangePrivate ( Point &i ) { i.m_i++; }

int main()
{
   Point sPoint;
   sPoint.PrintPrivate();
   ChangePrivate(sPoint);
   sPoint.PrintPrivate();
    // Output: 0
           1
}
```

Class members as friends

```
class B;

class A {
public:
   int Func1( B& b );

private:
   int Func2( B& b );
};

class B {
private:
int _b;

   // A::Func1 is a friend function to class B
   // so A::Func1 has access to all members of B
   friend int A::Func1( B& );
};

int A::Func1( B& b ) { return b._b; }   // OK
int A::Func2( B& b ) { return b._b; }   // C2248
```

Section 23.3: typename

1. When followed by a qualified name, `typename` specifies that it is the name of a type. This is often required in templates, in particular, when the nested name specifier is a dependent type other than the current instantiation. In this example, `std::decay<T>` depends on the template parameter `T`, so in order to name the nested type `type`, we need to prefix the entire qualified name with `typename`. For more deatils, see Where and why do I have to put the "template" and "typename" keywords?

```
template <class T>
auto decay_copy(T&& r) -> typename std::decay<T>::type;
```

2. Introduces a type parameter in the declaration of a template. In this context, it is interchangeable with `class`.

```
template <typename T>
const T& min(const T& x, const T& y) {
    return b < a ? b : a;
}
```

Version ≥ C++17

3. `typename` can also be used when declaring a template template parameter, preceding the name of the parameter, just like `class`.

```
template <template <class T> typename U>
void f() {
    U<int>::do_it();
    U<double>::do_it();
}
```

Section 23.4: explicit

1. When applied to a single-argument constructor, prevents that constructor from being used to perform implicit conversions.

```
class MyVector {
  public:
    explicit MyVector(uint64_t size);
};
MyVector v1(100);  // ok
uint64_t len1 = 100;
MyVector v2{len1}; // ok, len1 is uint64_t
int len2 = 100;
MyVector v3{len2}; // ill-formed, implicit conversion from int to uint64_t
```

Since C++11 introduced initializer lists, in C++11 and later, `explicit` can be applied to a constructor with any number of arguments, with the same meaning as in the single-argument case.

```
struct S {
    explicit S(int x, int y);
};
S f() {
    return {12, 34};  // ill-formed
    return S{12, 34}; // ok
}
```

Version ≥ C++11

2. When applied to a conversion function, prevents that conversion function from being used to perform implicit conversions.

```
class C {
    const int x;
  public:
    C(int x) : x(x) {}
    explicit operator int() { return x; }
};
C c(42);
int x = c;                   // ill-formed
int y = static_cast<int>(c); // ok; explicit conversion
```

Section 23.5: sizeof

A unary operator that yields the size in bytes of its operand, which may be either an expression or a type. If the operand is an expression, it is not evaluated. The size is a constant expression of type `std::size_t`.

If the operand is a type, it must be parenthesized.

- It is illegal to apply `sizeof` to a function type.
- It is illegal to apply `sizeof` to an incomplete type, including `void`.
- If sizeof is applied to a reference type `T&` or `T&&`, it is equivalent to `sizeof(T)`.
- When `sizeof` is applied to a class type, it yields the number of bytes in a complete object of that type, including any padding bytes in the middle or at the end. Therefore, a `sizeof` expression can never have a value of 0. See layout of object types for more details.

- The `char`, `signed char`, and `unsigned char` types have a size of 1. Conversely, a byte is defined to be the amount of memory required to store a `char` object. It does not necessarily mean 8 bits, as some systems have `char` objects longer than 8 bits.

If *expr* is an expression, `sizeof(`*expr*`)` is equivalent to `sizeof(T)` where `T` is the type of *expr.*

```
int a[100];
std::cout << "The number of bytes in `a` is: " << sizeof a;
memset(a, 0, sizeof a); // zeroes out the array
```

Version ≥ C++11

The `sizeof...` operator yields the number of elements in a parameter pack.

```
template <class... T>
void f(T&&...) {
    std::cout << "f was called with " << sizeof...(T) << " arguments\n";
}
```

Section 23.6: noexcept

Version ≥ C++11

1. A unary operator that determines whether the evaluation of its operand can propagate an exception. Note that the bodies of called functions are not examined, so `noexcept` can yield false negatives. The operand is not evaluated.

```
#include <iostream>
#include <stdexcept>
void foo() { throw std::runtime_error("oops"); }
void bar() {}
struct S {};
int main() {
    std::cout << noexcept(foo()) << '\n'; // prints 0
    std::cout << noexcept(bar()) << '\n'; // prints 0
    std::cout << noexcept(1 + 1) << '\n'; // prints 1
    std::cout << noexcept(S()) << '\n';   // prints 1
}
```

In this example, even though `bar()` can never throw an exception, `noexcept(bar())` is still false because the fact that `bar()` cannot propagate an exception has not been explicitly specified.

2. When declaring a function, specifies whether or not the function can propagate an exception. Alone, it declares that the function cannot propagate an exception. With a parenthesized argument, it declares that the function can or cannot propagate an exception depending on the truth value of the argument.

```
void f1() { throw std::runtime_error("oops"); }
void f2() noexcept(false) { throw std::runtime_error("oops"); }
void f3() {}
void f4() noexcept {}
void f5() noexcept(true) {}
void f6() noexcept {
    try {
        f1();
    } catch (const std::runtime_error&) {}
}
```

In this example, we have declared that `f4`, `f5`, and `f6` cannot propagate exceptions. (Although an exception can be thrown during execution of `f6`, it is caught and not allowed to propagate out of the function.) We have declared that `f2` may propagate an exception. When the `noexcept` specifier is omitted, it is equivalent to `noexcept(false)`, so we have implicitly declared that `f1` and `f3` may propagate exceptions, even though exceptions cannot actually be thrown during the execution of `f3`.

Version ≥ C++17

Whether or not a function is `noexcept` is part of the function's type: that is, in the example above, `f1`, `f2`, and `f3` have different types from `f4`, `f5`, and `f6`. Therefore, `noexcept` is also significant in function pointers, template arguments, and so on.

```
void g1() {}
void g2() noexcept {}
void (*p1)() noexcept = &g1; // ill-formed, since g1 is not noexcept
void (*p2)() noexcept = &g2; // ok; types match
void (*p3)() = &g1;          // ok; types match
void (*p4)() = &g2;          // ok; implicit conversion
```

Chapter 24: Returning several values from a function

There are many situations where it is useful to return several values from a function: for example, if you want to input an item and return the price and number in stock, this functionality could be useful. There are many ways to do this in C++, and most involve the STL. However, if you wish to avoid the STL for some reason, there are still several ways to do this, including `structs/classes` and `arrays`.

Section 24.1: Using std::tuple

Version ≥ C++11

The type `std::tuple` can bundle any number of values, potentially including values of different types, into a single return object:

```
std::tuple<int, int, int, int> foo(int a, int b) { // or auto (C++14)
   return std::make_tuple(a + b, a - b, a * b, a / b);
}
```

In C++17, a braced initializer list can be used:

Version ≥ C++17

```
std::tuple<int, int, int, int> foo(int a, int b)    {
    return {a + b, a - b, a * b, a / b};
}
```

Retrieving values from the returned `tuple` can be cumbersome, requiring the use of the `std::get` template function:

```
auto mrvs = foo(5, 12);
auto add = std::get<0>(mrvs);
auto sub = std::get<1>(mrvs);
auto mul = std::get<2>(mrvs);
auto div = std::get<3>(mrvs);
```

If the types can be declared before the function returns, then `std::tie` can be employed to unpack a `tuple` into existing variables:

```
int add, sub, mul, div;
std::tie(add, sub, mul, div) = foo(5, 12);
```

If one of the returned values is not needed, `std::ignore` can be used:

```
std::tie(add, sub, std::ignore, div) = foo(5, 12);
```

Version ≥ C++17

Structured bindings can be used to avoid `std::tie`:

```
auto [add, sub, mul, div] = foo(5,12);
```

If you want to return a tuple of lvalue references instead of a tuple of values, use `std::tie` in place of `std::make_tuple`.

```
std::tuple<int&, int&> minmax( int& a, int& b ) {
```

```
  if (b<a)
    return std::tie(b,a);
  else
    return std::tie(a,b);
}
```

which permits

```
void increase_least(int& a, int& b) {
  std::get<0>(minmax(a,b))++;
}
```

In some rare cases you'll use `std::forward_as_tuple` instead of `std::tie`; be careful if you do so, as temporaries may not last long enough to be consumed.

Section 24.2: Structured Bindings

Version ≥ C++17

C++17 introduces structured bindings, which makes it even easier to deal with multiple return types, as you do not need to rely upon `std::tie()` or do any manual tuple unpacking:

```
std::map<std::string, int> m;

// insert an element into the map and check if insertion succeeded
auto [iterator, success] = m.insert({"Hello", 42});

if (success) {
    // your code goes here
}

// iterate over all elements without having to use the cryptic 'first' and 'second' names
for (auto const& [key, value] : m) {
    std::cout << "The value for " << key << " is " << value << '\n';
}
```

Structured bindings can be used by default with `std::pair`, `std::tuple`, and any type whose non-static data members are all either public direct members or members of an unambiguous base class:

```
struct A { int x; };
struct B : A { int y; };
B foo();

// with structured bindings
const auto [x, y] = foo();

// equivalent code without structured bindings
const auto result = foo();
auto& x = result.x;
auto& y = result.y;
```

If you make your type "tuple-like" it will also automatically work with your type. A tuple-like is a type with appropriate `tuple_size`, `tuple_element` and `get` written:

```
namespace my_ns {
    struct my_type {
        int x;
```

```
        double d;
        std::string s;
    };
    struct my_type_view {
        my_type* ptr;
    };
}

namespace std {
    template<>
    struct tuple_size<my_ns::my_type_view> : std::integral_constant<std::size_t, 3>
    {};

    template<> struct tuple_element<my_ns::my_type_view, 0>{ using type = int; };
    template<> struct tuple_element<my_ns::my_type_view, 1>{ using type = double; };
    template<> struct tuple_element<my_ns::my_type_view, 2>{ using type = std::string; };
}

namespace my_ns {
    template<std::size_t I>
    decltype(auto) get(my_type_view const& v) {
        if constexpr (I == 0)
            return v.ptr->x;
        else if constexpr (I == 1)
            return v.ptr->d;
        else if constexpr (I == 2)
            return v.ptr->s;
        static_assert(I < 3, "Only 3 elements");
    }
}
```

now this works:

```
my_ns::my_type t{1, 3.14, "hello world"};

my_ns::my_type_view foo() {
    return {&t};
}

int main() {
    auto[x, d, s] = foo();
    std::cout << x << ',' << d << ',' << s << '\n';
}
```

Section 24.3: Using struct

A struct can be used to bundle multiple return values:

Version ≥ C++11

```
struct foo_return_type {
    int add;
    int sub;
    int mul;
    int div;
};

foo_return_type foo(int a, int b) {
    return {a + b, a - b, a * b, a / b};
}
```

```
auto calc = foo(5, 12);
```

Version < C++11

Instead of assignment to individual fields, a constructor can be used to simplify the constructing of returned values:

```
struct foo_return_type {
    int add;
    int sub;
    int mul;
    int div;
    foo_return_type(int add, int sub, int mul, int div)
    : add(add), sub(sub), mul(mul), div(div) {}
};

foo_return_type foo(int a, int b) {
     return foo_return_type(a + b, a - b, a * b, a / b);
}

foo_return_type calc = foo(5, 12);
```

The individual results returned by the function `foo()` can be retrieved by accessing the member variables of the `struct` `calc`:

```
std::cout << calc.add << ' ' << calc.sub << ' ' << calc.mul << ' ' << calc.div << '\n';
```

Output:

> 17 -7 60 0

Note: When using a `struct`, the returned values are grouped together in a single object and accessible using meaningful names. This also helps to reduce the number of extraneous variables created in the scope of the returned values.

Version ≥ C++17

In order to unpack a `struct` returned from a function, structured bindings can be used. This places the out-parameters on an even footing with the in-parameters:

```
int a=5, b=12;
auto[add, sub, mul, div] = foo(a, b);
std::cout << add << ' ' << sub << ' ' << mul << ' ' << div << '\n';
```

The output of this code is identical to that above. The `struct` is still used to return the values from the function. This permits you do deal with the fields individually.

Section 24.4: Using Output Parameters

Parameters can be used for returning one or more values; those parameters are required to be non-`const` pointers or references.

References:

```
void calculate(int a, int b, int& c, int& d, int& e, int& f) {
    c = a + b;
    d = a - b;
```

```
    e = a * b;
    f = a / b;
}
```

Pointers:

```
void calculate(int a, int b, int* c, int* d, int* e, int* f) {
    *c = a + b;
    *d = a - b;
    *e = a * b;
    *f = a / b;
}
```

Some libraries or frameworks use an empty 'OUT' `#define` to make it abundantly obvious which parameters are output parameters in the function signature. This has no functional impact, and will be compiled out, but makes the function signature a bit clearer;

```
#define OUT

void calculate(int a, int b, OUT int& c) {
    c = a + b;
}
```

Section 24.5: Using a Function Object Consumer

We can provide a consumer that will be called with the multiple relevant values:

Version ≥ C++11

```
template <class F>
void foo(int a, int b, F consumer) {
    consumer(a + b, a - b, a * b, a / b);
}

// use is simple... ignoring some results is possible as well
foo(5, 12, [](int sum, int , int , int ){
    std::cout << "sum is " << sum << '\n';
});
```

This is known as "continuation passing style".

You can adapt a function returning a tuple into a continuation passing style function via:

Version ≥ C++17

```
template<class Tuple>
struct continuation {
  Tuple t;
  template<class F>
  decltype(auto) operator->*(F&& f)&&{
    return std::apply( std::forward<F>(f), std::move(t) );
  }
};
std::tuple<int,int,int,int> foo(int a, int b);

continuation(foo(5,12))->*[](int sum, auto&&...) {
  std::cout << "sum is " << sum << '\n';
};
```

with more complex versions being writable in C++14 or C++11.

Section 24.6: Using std::pair

The struct template std::pair can bundle together *exactly* two return values, of any two types:

```
#include <utility>
std::pair<int, int> foo(int a, int b) {
    return std::make_pair(a+b, a-b);
}
```

With C++11 or later, an initializer list can be used instead of `std::make_pair`:

Version ≥ C++11

```
#include <utility>
std::pair<int, int> foo(int a, int b) {
    return {a+b, a-b};
}
```

The individual values of the returned `std::pair` can be retrieved by using the pair's `first` and `second` member objects:

```
std::pair<int, int> mrvs = foo(5, 12);
std::cout << mrvs.first + mrvs.second << std::endl;
```

Output:

> 10

Section 24.7: Using std::array

Version ≥ C++11

The container `std::array` can bundle together a fixed number of return values. This number has to be known at compile-time and all return values have to be of the same type:

```
std::array<int, 4> bar(int a, int b) {
    return { a + b, a - b, a * b, a / b };
}
```

This replaces c style arrays of the form `int bar[4]`. The advantage being that various `c++` std functions can now be used on it. It also provides useful member functions like `at` which is a safe member access function with bound checking, and `size` which allows you to return the size of the array without calculation.

Section 24.8: Using Output Iterator

Several values of the same type can be returned by passing an output iterator to the function. This is particularly common for generic functions (like the algorithms of the standard library).

Example:

```
template<typename Incrementable, typename OutputIterator>
void generate_sequence(Incrementable from, Incrementable to, OutputIterator output) {
    for (Incrementable k = from; k != to; ++k)
        *output++ = k;
```

```
}
```

Example usage:

```
std::vector<int> digits;
generate_sequence(0, 10, std::back_inserter(digits));
// digits now contains {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

Section 24.9: Using std::vector

A `std::vector` can be useful for returning a dynamic number of variables of the same type. The following example uses `int` as data type, but a `std::vector` can hold any type that is trivially copyable:

```
#include <vector>
#include <iostream>

// the following function returns all integers between and including 'a' and 'b' in a vector
// (the function can return up to std::vector::max_size elements with the vector, given that
// the system's main memory can hold that many items)
std::vector<int> fillVectorFrom(int a, int b) {
    std::vector<int> temp;
    for (int i = a; i <= b; i++) {
        temp.push_back(i);
    }
    return temp;
}

int main() {
    // assigns the filled vector created inside the function to the new vector 'v'
    std::vector<int> v = fillVectorFrom(1, 10);

    // prints "1 2 3 4 5 6 7 8 9 10 "
    for (int i = 0; i < v.size(); i++) {
        std::cout << v[i] << " ";
    }
    std::cout << std::endl;
    return 0;
}
```

Chapter 25: Polymorphism

Section 25.1: Define polymorphic classes

The typical example is an abstract shape class, that can then be derived into squares, circles, and other concrete shapes.

The parent class:

Let's start with the polymorphic class:

```
class Shape {
public:
    virtual ~Shape() = default;
    virtual double get_surface() const = 0;
    virtual void describe_object() const { std::cout << "this is a shape" << std::endl; }

    double get_doubled_surface() const { return 2 * get_surface(); }
};
```

How to read this definition ?

- You can define polymorphic behavior by introduced member functions with the keyword `virtual`. Here `get_surface()` and `describe_object()` will obviously be implemented differently for a square than for a circle. When the function is invoked on an object, function corresponding to the real class of the object will be determined at runtime.
- It makes no sense to define `get_surface()` for an abstract shape. This is why the function is followed by `= 0`. This means that the function is *pure virtual function*.
- A polymorphic class should always define a virtual destructor.
- You may define non virtual member functions. When these function will be invoked for an object, the function will be chosen depending on the class used at compile-time. Here `get_double_surface()` is defined in this way.
- A class that contains at least one pure virtual function is an abstract class. Abstract classes cannot be instantiated. You may only have pointers or references of an abstract class type.

Derived classes

Once a polymorphic base class is defined you can derive it. For example:

```
class Square : public Shape {
    Point top_left;
    double side_length;
public:
    Square (const Point& top_left, double side)
       : top_left(top_left), side_length(side_length) {}

    double get_surface() override { return side_length * side_length; }
    void describe_object() override {
        std::cout << "this is a square starting at " << top_left.x << ", " << top_left.y
                  << " with a length of " << side_length << std::endl;
    }
};
```

Some explanations:

- You can define or override any of the virtual functions of the parent class. The fact that a function was virtual in the parent class makes it virtual in the derived class. No need to tell the compiler the keyword `virtual` again. But it's recommended to add the keyword `override` at the end of the function declaration, in order to prevent subtle bugs caused by unnoticed variations in the function signature.
- If all the pure virtual functions of the parent class are defined you can instantiate objects for this class, else it will also become an abstract class.
- You are not obliged to override all the virtual functions. You can keep the version of the parent if it suits your need.

Example of instantiation

```
int main() {

    Square square(Point(10.0, 0.0), 6); // we know it's a square, the compiler also
    square.describe_object();
    std::cout << "Surface: " << square.get_surface() << std::endl;

    Circle circle(Point(0.0, 0.0), 5);

    Shape *ps = nullptr;  // we don't know yet the real type of the object
    ps = &circle;         // it's a circle, but it could as well be a square
    ps->describe_object();
    std::cout << "Surface: " << ps->get_surface() << std::endl;
}
```

Section 25.2: Safe downcasting

Suppose that you have a pointer to an object of a polymorphic class:

```
Shape *ps;                       // see example on defining a polymorphic class
ps =  get_a_new_random_shape();  // if you don't have such a function yet, you
                                 // could just write ps = new Square(0.0,0.0, 5);
```

a downcast would be to cast from a general polymorphic `Shape` down to one of its derived and more specific shape like `Square` or `Circle`.

Why to downcast ?

Most of the time, you would not need to know which is the real type of the object, as the virtual functions allow you to manipulate your object independently of its type:

```
std::cout << "Surface: " << ps->get_surface() << std::endl;
```

If you don't need any downcast, your design would be perfect.

However, you may need sometimes to downcast. A typical example is when you want to invoke a non virtual function that exist only for the child class.

Consider for example circles. Only circles have a diameter. So the class would be defined as :

```
class Circle: public Shape { // for Shape, see example on defining a polymorphic class
    Point center;
    double radius;
public:
```

```
    Circle (const Point& center, double radius)
        : center(center), radius(radius) {}

    double get_surface() const override { return r * r * M_PI; }

    // this is only for circles. Makes no sense for other shapes
    double get_diameter() const { return 2 * r; }
};
```

The `get_diameter()` member function only exist for circles. It was not defined for a `Shape` object:

```
Shape* ps = get_any_shape();
ps->get_diameter(); // OUCH !!! Compilation error
```

How to downcast ?

If you'd know for sure that `ps` points to a circle you could opt for a `static_cast`:

```
std::cout << "Diameter: " << static_cast<Circle*>(ps)->get_diameter() << std::endl;
```

This will do the trick. But it's very risky: if `ps` appears to by anything else than a `Circle` the behavior of your code will be undefined.

So rather than playing Russian roulette, you should safely use a `dynamic_cast`. This is specifically for polymorphic classes :

```
int main() {
    Circle circle(Point(0.0, 0.0), 10);
    Shape &shape = circle;

    std::cout << "The shape has a surface of " << shape.get_surface() << std::endl;

    //shape.get_diameter();   // OUCH !!! Compilation error

    Circle *pc = dynamic_cast<Circle*>(&shape); // will be nullptr if ps wasn't a circle
    if (pc)
        std::cout << "The shape is a circle of diameter " << pc->get_diameter() << std::endl;
    else
        std::cout << "The shape isn't a circle !" << std::endl;
}
```

Note that `dynamic_cast` is not possible on a class that is not polymorphic. You'd need at least one virtual function in the class or its parents to be able to use it.

Section 25.3: Polymorphism & Destructors

If a class is intended to be used polymorphically, with derived instances being stored as base pointers/references, its base class' destructor should be either `virtual` or `protected`. In the former case, this will cause object destruction to check the `vtable`, automatically calling the correct destructor based on the dynamic type. In the latter case, destroying the object through a base class pointer/reference is disabled, and the object can only be deleted when explicitly treated as its actual type.

```
struct VirtualDestructor {
    virtual ~VirtualDestructor() = default;
};

struct VirtualDerived : VirtualDestructor {};
```

```cpp
struct ProtectedDestructor {
  protected:
    ~ProtectedDestructor() = default;
};

struct ProtectedDerived : ProtectedDestructor {
    ~ProtectedDerived() = default;
};

// ...

VirtualDestructor* vd = new VirtualDerived;
delete vd; // Looks up VirtualDestructor::~VirtualDestructor() in vtable, sees it's
           // VirtualDerived::~VirtualDerived(), calls that.

ProtectedDestructor* pd = new ProtectedDerived;
delete pd; // Error: ProtectedDestructor::~ProtectedDestructor() is protected.
delete static_cast<ProtectedDerived*>(pd); // Good.
```

Both of these practices guarantee that the derived class' destructor will always be called on derived class instances, preventing memory leaks.

Chapter 26: References

Section 26.1: Defining a reference

References behaves similarly, but not entirely like const pointers. A reference is defined by suffixing an ampersand `&` to a type name.

```
int i = 10;
int &refi = i;
```

Here, `refi` is a reference bound to `i`.
References abstracts the semantics of pointers, acting like an alias to the underlying object:

```
refi = 20; // i = 20;
```

You can also define multiple references in a single definition:

```
int i = 10, j = 20;
int &refi = i, &refj = j;

// Common pitfall :
// int& refi = i, k = j;
// refi will be of type int&.
// though, k will be of type int, not int&!
```

References **must** be initialized correctly at the time of definition, and cannot be modified afterwards. The following piece of codes causes a compile error:

```
int &i; // error: declaration of reference variable 'i' requires an initializer
```

You also cannot bind directly a reference to `nullptr`, unlike pointers:

```
int *const ptri = nullptr;
int &refi = nullptr; // error: non-const lvalue reference to type 'int' cannot bind to a temporary
of type 'nullptr_t'
```

Chapter 27: Value and Reference Semantics

Section 27.1: Definitions

A type has value semantics if the object's observable state is functionally distinct from all other objects of that type. This means that if you copy an object, you have a new object, and modifications of the new object will not be in any way visible from the old object.

Most basic C++ types have value semantics:

```
int i = 5;
int j = i; //Copied
j += 20;
std::cout << i; //Prints 5; i is unaffected by changes to j.
```

Most standard-library defined types have value semantics too:

```
std::vector<int> v1(5, 12); //array of 5 values, 12 in each.
std::vector<int> v2 = v1; //Copies the vector.
v2[3] = 6; v2[4] = 9;
std::cout << v1[3] << " " << v1[4]; //Writes "12 12", since v1 is unchanged.
```

A type is said to have reference semantics if an instance of that type can share its observable state with another object (external to it), such that manipulating one object will cause the state to change within another object.

C++ pointers have value semantics with regard to which object they point to, but they have reference semantics with regard to the *state* of the object they point to:

```
int *pi = new int(4);
int *pi2 = pi;
pi = new int(16);
assert(pi2 != pi); //Will always pass.

int *pj = pi;
*pj += 5;
std::cout << *pi; //Writes 9, since `pi` and `pj` reference the same object.
```

C++ references have reference semantics as well.

Section 27.2: Deep copying and move support

If a type wishes to have value semantics, and it needs to store objects that are dynamically allocated, then on copy operations, the type will need to allocate new copies of those objects. It must also do this for copy assignment.

This kind of copying is called a "deep copy". It effectively takes what would have otherwise been reference semantics and turns it into value semantics:

```
struct Inner {int i;};

const int NUM_INNER = 5;
class Value
{
private:
  Inner *array_; //Normally has reference semantics.
```

```
public:
  Value() : array_(new Inner[NUM_INNER]){}

  ~Value() {delete[] array_;}

  Value(const Value &val) : array_(new Inner[NUM_INNER])
  {
    for(int i = 0; i < NUM_INNER; ++i)
      array_[i] = val.array_[i];
  }

  Value &operator=(const Value &val)
  {
    for(int i = 0; i < NUM_INNER; ++i)
      array_[i] = val.array_[i];
    return *this;
  }
};
```

Version ≥ C++11

Move semantics allow a type like `Value` to avoid truly copying its referenced data. If the user uses the value in a way that provokes a move, the "copied" from object can be left empty of the data it referenced:

```
struct Inner {int i;};

constexpr auto NUM_INNER = 5;
class Value
{
private:
  Inner *array_; //Normally has reference semantics.

public:
  Value() : array_(new Inner[NUM_INNER]){}

  //OK to delete even if nullptr
  ~Value() {delete[] array_;}

  Value(const Value &val) : array_(new Inner[NUM_INNER])
  {
    for(int i = 0; i < NUM_INNER; ++i)
      array_[i] = val.array_[i];
  }

  Value &operator=(const Value &val)
  {
    for(int i = 0; i < NUM_INNER; ++i)
      array_[i] = val.array_[i];
    return *this;
  }

  //Movement means no memory allocation.
  //Cannot throw exceptions.
  Value(Value &&val) noexcept : array_(val.array_)
  {
    //We've stolen the old value.
    val.array_ = nullptr;
  }

  //Cannot throw exceptions.
  Value &operator=(Value &&val) noexcept
  {
```

```
    //Clever trick. Since `val` is going to be destroyed soon anyway,
    //we swap his data with ours. His destructor will destroy our data.
    std::swap(array_, val.array_);
  }
};
```

Indeed, we can even make such a type non-copyable, if we want to forbid deep copies while still allowing the object to be moved around.

```
struct Inner {int i;};

constexpr auto NUM_INNER = 5;
class Value
{
private:
  Inner *array_; //Normally has reference semantics.

public:
  Value() : array_(new Inner[NUM_INNER]){}

  //OK to delete even if nullptr
  ~Value() {delete[] array_;}

  Value(const Value &val) = delete;
  Value &operator=(const Value &val) = delete;

  //Movement means no memory allocation.
  //Cannot throw exceptions.
  Value(Value &&val) noexcept : array_(val.array_)
  {
    //We've stolen the old value.
    val.array_ = nullptr;
  }

  //Cannot throw exceptions.
  Value &operator=(Value &&val) noexcept
  {
    //Clever trick. Since `val` is going to be destroyed soon anyway,
    //we swap his data with ours. His destructor will destroy our data.
    std::swap(array_, val.array_);
  }
};
```

We can even apply the Rule of Zero, through the use of `unique_ptr`:

```
struct Inner {int i;};

constexpr auto NUM_INNER = 5;
class Value
{
private:
  unique_ptr<Inner []>array_; //Move-only type.

public:
  Value() : array_(new Inner[NUM_INNER]){}

  //No need to explicitly delete. Or even declare.
  ~Value() = default; {delete[] array_;}

  //No need to explicitly delete. Or even declare.
  Value(const Value &val) = default;
```

```
    Value &operator=(const Value &val) = default;

    //Will perform an element-wise move.
    Value(Value &&val) noexcept = default;

    //Will perform an element-wise move.
    Value &operator=(Value &&val) noexcept = default;
};
```

Chapter 28: C++ function "call by value" vs. "call by reference"

The scope of this section is to explain the differences in theory and implementation for what happens with the parameters of a function upon calling.

In detail the parameters can be seen as variables before the function call and inside the function, where the visible behaviour and accessibility to these variables differs with the method used to hand them over.

Additionally, the reusability of variables and their respective values after the function call also is explained by this topic.

Section 28.1: Call by value

Upon calling a function there are new elements created on the program stack. These include some information about the function and also space (memory locations) for the parameters and the return value.

When handing over a parameter to a function the value of the used variable (or literal) is copied into the memory location of the function parameter. This implies that now there a two memory locations with the same value. Inside of the function we only work on the parameter memory location.

After leaving the function the memory on the program stack is popped (removed) which erases all data of the function call, including the memory location of the parameters we used inside. Thus, the values changed inside the function do not affect the outside variables values.

```
int func(int f, int b) {
  //new variables are created and values from the outside copied
  //f has a value of 0
  //inner_b has a value of 1
  f = 1;
  //f has a value of 1
  b = 2;
  //inner_b has a value of 2
  return f+b;
}

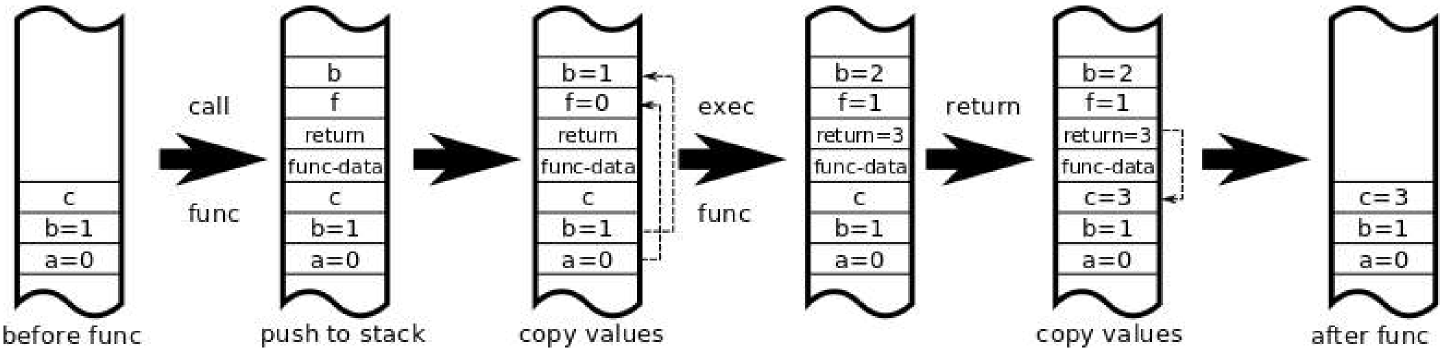
int main(void) {
  int a = 0;
  int b = 1; //outer_b
  int c;

  c = func(a,b);
  //the return value is copied to c

  //a has a value of 0
  //outer_b has a value of 1   <--- outer_b and inner_b are different variables
  //c has a value of 3
}
```

In this code we create variables inside the main function. These get assigned values. Upon calling the functions there are two new variables created: `f` and `inner_b` where `b` shares the name with the outer variable it does not share the memory location. The behaviour of `a<->f` and `b<->b` is identical.

The following graphic symbolizes what is happening on the stack and why there is no change in varibale `b`. The graphic is not fully accurate but emphazises the example.



It is called "call by value" because we do not hand over the variables but only the values of these variables.

Chapter 29: Copying vs Assignment

rhs	**Right Hand Side of the equality for both copy and assignment constructors. For example the assignment constructor : MyClass operator=(MyClass& rhs);**

Placeholder Placeholder

Section 29.1: Assignment Operator

The Assignment Operator is when you replace the data with an already existing(previously initialized) object with some other object's data. Lets take this as an example:

```
// Assignment Operator
#include <iostream>
#include <string>

using std::cout;
using std::endl;

class Foo
{
  public:
    Foo(int data)
    {
        this->data = data;
    }
    ~Foo(){};
    Foo& operator=(const Foo& rhs)
    {
            data = rhs.data;
            return *this;
    }

    int data;
};

int main()
{
   Foo foo(2); //Foo(int data) called
   Foo foo2(42);
   foo = foo2; // Assignment Operator Called
   cout << foo.data << endl; //Prints 42
}
```

You can see here I call the assignment operator when I already initialized the `foo` object. Then later I assign `foo2` to `foo` . All the changes to appear when you call that equal sign operator is defined in your `operator=` function. You can see a runnable output here: http://cpp.sh/3qtbm

Section 29.2: Copy Constructor

Copy constructor on the other hand , is the complete opposite of the Assignment Constructor. This time, it is used to initialize an already nonexistent(or non-previously initialized) object. This means it copies all the data from the object you are assigning it to , without actually initializing the object that is being copied onto. Now Let's take a look at the same code as before but modify the assignment constructor to be a copy constructor :

```
// Copy Constructor
#include <iostream>
#include <string>
```

```
using std::cout;
using std::endl;

class Foo
{
  public:
    Foo(int data)
    {
        this->data = data;
    }
    ~Foo(){};
    Foo(const Foo& rhs)
    {
            data = rhs.data;
    }

    int data;
};

int main()
{
   Foo foo(2); //Foo(int data) called
   Foo foo2 = foo; // Copy Constructor called
   cout << foo2.data << endl;
}
```

You can see here `Foo foo2 = foo;` in the main function I immediately assign the object before actually initializing it, which as said before means it's a copy constructor. And notice that I didn't need to pass the parameter int for the `foo2` object since I automatically pulled the previous data from the object foo. Here is an example output : http://cpp.sh/5iu7

Section 29.3: Copy Constructor Vs Assignment Constructor

Ok we have briefly looked over what the copy constructor and assignment constructor are above and gave examples of each now let's see both of them in the same code. This code will be similar as above two. Let's take this :

```
// Copy vs Assignment Constructor
#include <iostream>
#include <string>

using std::cout;
using std::endl;

class Foo
{
  public:
    Foo(int data)
    {
        this->data = data;
    }
    ~Foo(){};
    Foo(const Foo& rhs)
    {
            data = rhs.data;
    }

    Foo& operator=(const Foo& rhs)
    {
        data = rhs.data;
```

```
        return *this;
    }

    int data;
};

int main()
{
   Foo foo(2); //Foo(int data) / Normal Constructor called
   Foo foo2 = foo; //Copy Constructor Called
   cout << foo2.data << endl;

   Foo foo3(42);
   foo3=foo; //Assignment Constructor Called
   cout << foo3.data << endl;
}
```

Output:

```
2
2
```

Here you can see we first call the copy constructor by executing the line `Foo foo2 = foo;` . Since we didn't initialize it previously. And then next we call the assignment operator on foo3 since it was already initialized `foo3=foo`;

Chapter 30: Pointers

A pointer is an address that refers to a location in memory. They're commonly used to allow functions or data structures to know of and modify memory without having to copy the memory referred to. Pointers are usable with both primitive (built-in) or user-defined types.

Pointers make use of the "dereference" `*`, "address of" `&`, and "arrow" `->` operators. The '*' and '->' operators are used to access the memory being pointed at, and the `&` operator is used to get an address in memory.

Section 30.1: Pointer Operations

There are two operators for pointers: Address-of operator (&): Returns the memory address of its operand. Contents-of (Dereference) operator(*): Returns the value of the variable located at the address specified by its operator.

```
int var = 20;
int *ptr;
ptr = &var;

cout << var << endl;
//Outputs 20 (The value of var)

cout << ptr << endl;
//Outputs 0x234f119 (var's memory location)

cout << *ptr << endl;
//Outputs 20(The value of the variable stored in the pointer ptr
```

The asterisk (*) is used in declaring a pointer for simple purpose of indicating that it is a pointer. Don't confuse this with the **dereference** operator, which is used to obtain the value located at the specified address. They are simply two different things represented with the same sign.

Section 30.2: Pointer basics

`Version < C++11`

Note: in all the following, the existence of the C++11 constant `nullptr` is assumed. For earlier versions, replace `nullptr` with `NULL`, the constant that used to play a similar role.

Creating a pointer variable

A pointer variable can be created using the specific `*` syntax, e.g. `int *pointer_to_int;`.
When a variable is of a *pointer type* (`int *`), it just contains a memory address. The memory address is the location at which data of the *underlying type* (`int`) is stored.

The difference is clear when comparing the size of a variable with the size of a pointer to the same type:

```
// Declare a struct type `big_struct` that contains
// three long long ints.
typedef struct {
    long long int foo1;
    long long int foo2;
    long long int foo3;
} big_struct;

// Create a variable `bar` of type `big_struct`
```

```cpp
big_struct bar;
// Create a variable `p_bar` of type `pointer to big_struct`.
// Initialize it to `nullptr` (a null pointer).
big_struct *p_bar0 = nullptr;

// Print the size of `bar`
std::cout << "sizeof(bar) = " << sizeof(bar) << std::endl;
// Print the size of `p_bar`.
std::cout << "sizeof(p_bar0) = " << sizeof(p_bar0) << std::endl;

/* Produces:
    sizeof(bar) = 24
    sizeof(p_bar0) = 8
*/
```

Taking the address of another variable

Pointers can be assigned between each other just as normal variables; in this case, it is the **memory address** that is copied from one pointer to another, **not the actual data** that a pointer points to.
Moreover, they can take the value `nullptr` which represents a null memory location. A pointer equal to `nullptr` contains an invalid memory location and hence it does not refer to valid data.

You can get the memory address of a variable of a given type by prefixing the variable with the *address of* operator `&`. The value returned by `&` is a pointer to the underlying type which contains the memory address of the variable (which is valid data **as long as the variable does not go out of scope**).

```cpp
// Copy `p_bar0` into `p_bar_1`.
big_struct *p_bar1 = p_bar0;

// Take the address of `bar` into `p_bar_2`
big_struct *p_bar2 = &bar;

// p_bar1 is now nullptr, p_bar2 is &bar.

p_bar0 = p_bar2;

// p_bar0 is now &bar.

p_bar2 = nullptr;

// p_bar0 == &bar
// p_bar1 == nullptr
// p_bar2 == nullptr
```

In contrast with references:

- assigning two pointers does not overwrite the memory that the assigned pointer refers to;
- pointers can be null.
- the *address of* operator is required explicitly.

Accessing the content of a pointer

As taking an address requires `&`, as well accessing the content requires the usage of the *dereference operator* `*`, as a prefix. When a pointer is dereferenced, it becomes a variable of the underlying type (actually, a reference to it). It can then be read and modified, if not `const`.

```cpp
(*p_bar0).foo1 = 5;

// `p_bar0` points to `bar`. This prints 5.
```

```
std::cout << "bar.foo1 = " << bar.foo1 << std::endl;

// Assign the value pointed to by `p_bar0` to `baz`.
big_struct baz;
baz = *p_bar0;

// Now `baz` contains a copy of the data pointed to by `p_bar0`.
// Indeed, it contains a copy of `bar`.

// Prints 5 as well
std::cout << "baz.foo1 = " << baz.foo1 << std::endl;
```

The combination of `*` and the operator `.` is abbreviated by `->`:

```
std::cout << "bar.foo1 = " << (*p_bar0).foo1 << std::endl; // Prints 5
std::cout << "bar.foo1 = " <<  p_bar0->foo1  << std::endl; // Prints 5
```

Dereferencing invalid pointers

When dereferencing a pointer, you should make sure it points to valid data. Dereferencing an invalid pointer (or a null pointer) can lead to memory access violation, or to read or write garbage data.

```
big_struct *never_do_this() {
   // This is a local variable. Outside `never_do_this` it doesn't exist.
   big_struct retval;
   retval.foo1 = 11;
   // This returns the address of `retval`.
   return &retval;
   // `retval` is destroyed and any code using the value returned
   // by `never_do_this` has a pointer to a memory location that
   // contains garbage data (or is inaccessible).
}
```

In such scenario, `g++` and `clang++` correctly issue the warnings:

```
(Clang) warning: address of stack memory associated with local variable 'retval' returned [-
Wreturn-stack-address]
(Gcc)   warning: address of local variable ‘retval’ returned [-Wreturn-local-addr]
```

Hence, care must be taken when pointers are arguments of functions, as they could be null:

```
void naive_code(big_struct *ptr_big_struct) {
    // ... some code which doesn't check if `ptr_big_struct` is valid.
    ptr_big_struct->foo1 = 12;
}

// Segmentation fault.
naive_code(nullptr);
```

Section 30.3: Pointer Arithmetic

Increment / Decrement

A pointer can be incremented or decremented (prefix and postfix). Incrementing a pointer advances the pointer value to the element in the array one element past the currently pointed to element. Decrementing a pointer moves it to the previous element in the array.

Pointer arithmetic is not permitted if the type that the pointer points to is not complete. `void` is always an incomplete type.

```
char* str = new char[10]; // str = 0x010
++str;                    // str = 0x011  in this case sizeof(char) = 1 byte

int* arr = new int[10];   // arr = 0x00100
++arr;                    // arr = 0x00104 if sizeof(int) = 4 bytes

void* ptr = (void*)new char[10];
++ptr;    // void is incomplete.
```

If a pointer to the end element is incremented, then the pointer points to one element past the end of the array. Such a pointer cannot be dereferenced, but it can be decremented.

Incrementing a pointer to the one-past-the-end element in the array, or decrementing a pointer to the first element in an array yields undefined behavior.

A pointer to a non-array object can be treated, for the purposes of pointer arithmetic, as though it were an array of size 1.

Addition / Subtraction

Integer values can be added to pointers; they act as incrementing, but by a specific number rather than by 1. Integer values can be subtracted from pointers as well, acting as pointer decrementing. As with incrementing/decrementing, the pointer must point to a complete type.

```
char* str = new char[10];  // str = 0x010
str += 2;                  // str = 0x010 + 2 * sizeof(char) = 0x012

int* arr = new int[10];    // arr = 0x100
arr += 2;                  // arr = 0x100 + 2 * sizeof(int) = 0x108, assuming sizeof(int) == 4.
```

Pointer Differencing

The difference between two pointers to the same type can be computed. The two pointers must be within the same array object; otherwise undefined behavior results.

Given two pointers `P` and `Q` in the same array, if `P` is the `i`th element in the array, and `Q` is the `j`th element, then `P - Q` shall be `i - j`. The type of the result is `std::ptrdiff_t`, from **`<cstddef>`**.

```
char* start = new char[10];  // str = 0x010
char* test = &start[5];
std::ptrdiff_t diff = test - start; //Equal to 5.
std::ptrdiff_t diff = start - test; //Equal to -5; ptrdiff_t is signed.
```

Chapter 31: Pointers to members

Section 31.1: Pointers to static member functions

A `static` member function is just like an ordinary C/C++ function, except with scope:

- It is inside a `class`, so it needs its name decorated with the class name;
- It has accessibility, with `public`, `protected` or `private`.

So, if you have access to the `static` member function and decorate it correctly, then you can point to the function like any normal function outside a `class`:

```
typedef int Fn(int); // Fn is a type-of function that accepts an int and returns an int

// Note that MyFn() is of type 'Fn'
int MyFn(int i) { return 2*i; }

class Class {
public:
    // Note that Static() is of type 'Fn'
    static int Static(int i) { return 3*i; }
}; // Class

int main() {
    Fn *fn;    // fn is a pointer to a type-of Fn

    fn = &MyFn;          // Point to one function
    fn(3);               // Call it
    fn = &Class::Static; // Point to the other function
    fn(4);               // Call it
 } // main()
```

Section 31.2: Pointers to member functions

To access a member function of a class, you need to have a "handle" to the particular instance, as either the instance itself, or a pointer or reference to it. Given a class instance, you can point to various of its members with a pointer-to-member, IF you get the syntax correct! Of course, the pointer has to be declared to be of the same type as what you are pointing to...

```
typedef int Fn(int); // Fn is a type-of function that accepts an int and returns an int

class Class {
public:
    // Note that A() is of type 'Fn'
    int A(int a) { return 2*a; }
    // Note that B() is of type 'Fn'
    int B(int b) { return 3*b; }
}; // Class

int main() {
    Class c;          // Need a Class instance to play with
    Class *p = &c;    // Need a Class pointer to play with

    Fn Class::*fn;    // fn is a pointer to a type-of Fn within Class

    fn = &Class::A;   // fn now points to A within any Class
    (c.*fn)(5);       // Pass 5 to c's function A (via fn)
```

```
    fn = &Class::B;    // fn now points to B within any Class
    (p->*fn)(6);       // Pass 6 to c's (via p) function B (via fn)
} // main()
```

Unlike pointers to member variables (in the previous example), the association between the class instance and the member pointer need to be bound tightly together with parentheses, which looks a little strange (as though the `.*` and `->*` aren't strange enough!)

Section 31.3: Pointers to member variables

To access a member of a `class`, you need to have a "handle" to the particular instance, as either the instance itself, or a pointer or reference to it. Given a `class` instance, you can point to various of its members with a pointer-to-member, IF you get the syntax correct! Of course, the pointer has to be declared to be of the same type as what you are pointing to...

```
class Class {
public:
    int x, y, z;
    char m, n, o;
}; // Class

int x;  // Global variable

int main() {
    Class c;        // Need a Class instance to play with
    Class *p = &c;  // Need a Class pointer to play with

    int *p_i;       // Pointer to an int

    p_i = &x;       // Now pointing to x
    p_i = &c.x;     // Now pointing to c's x

    int Class::*p_C_i; // Pointer to an int within Class

    p_C_i = &Class::x; // Point to x within any Class
    int i = c.*p_C_i;  // Use p_c_i to fetch x from c's instance
    p_C_i = &Class::y; // Point to y within any Class
    i = c.*p_C_i;      // Use p_c_i to fetch y from c's instance

    p_C_i = &Class::m; // ERROR! m is a char, not an int!

    char Class::*p_C_c = &Class::m; // That's better...
} // main()
```

The syntax of pointer-to-member requires some extra syntactic elements:

- To define the type of the pointer, you need to mention the base type, as well as the fact that it is inside a class: `int Class::*ptr;`.
- If you have a class or reference and want to use it with a pointer-to-member, you need to use the `.*` operator (akin to the `.` operator).
- If you have a pointer to a class and want to use it with a pointer-to-member, you need to use the `->*` operator (akin to the `->` operator).

Section 31.4: Pointers to static member variables

A `static` member variable is just like an ordinary C/C++ variable, except with scope:

- It is inside a `class`, so it needs its name decorated with the class name;
- It has accessibility, with `public`, `protected` or `private`.

So, if you have access to the `static` member variable and decorate it correctly, then you can point to the variable like any normal variable outside a `class`:

```
class Class {
public:
    static int i;
}; // Class

int Class::i = 1; // Define the value of i (and where it's stored!)

int j = 2;   // Just another global variable

int main() {
    int k = 3; // Local variable

    int *p;

    p = &k;   // Point to k
    *p = 2;   // Modify it
    p = &j;   // Point to j
    *p = 3;   // Modify it
    p = &Class::i; // Point to Class::i
    *p = 4;   // Modify it
} // main()
```

Chapter 32: The This Pointer

Section 32.1: this Pointer

All non-static member functions have a hidden parameter, a pointer to an instance of the class, named `this`; this parameter is silently inserted at the beginning of the parameter list, and handled entirely by the compiler. When a member of the class is accessed inside a member function, it is silently accessed through `this`; this allows the compiler to use a single non-static member function for all instances, and allows a member function to call other member functions polymorphically.

```
struct ThisPointer {
    int i;

    ThisPointer(int ii);

    virtual void func();

    int  get_i() const;
    void set_i(int ii);
};
ThisPointer::ThisPointer(int ii) : i(ii) {}
// Compiler rewrites as:
ThisPointer::ThisPointer(int ii) : this->i(ii) {}
// Constructor is responsible for turning allocated memory into 'this'.
// As the constructor is responsible for creating the object, 'this' will not be "fully"
// valid until the instance is fully constructed.

/* virtual */ void ThisPointer::func() {
    if (some_external_condition) {
        set_i(182);
    } else {
        i = 218;
    }
}
// Compiler rewrites as:
/* virtual */ void ThisPointer::func(ThisPointer* this) {
    if (some_external_condition) {
        this->set_i(182);
    } else {
        this->i = 218;
    }
}

int  ThisPointer::get_i() const { return i; }
// Compiler rewrites as:
int  ThisPointer::get_i(const ThisPointer* this) { return this->i; }

void ThisPointer::set_i(int ii) { i = ii; }
// Compiler rewrites as:
void ThisPointer::set_i(ThisPointer* this, int ii) { this->i = ii; }
```

In a constructor, `this` can safely be used to (implicitly or explicitly) access any field that has already been initialised, or any field in a parent class; conversely, (implicitly or explicitly) accessing any fields that haven't yet been initialised, or any fields in a derived class, is unsafe (due to the derived class not yet being constructed, and thus its fields neither being initialised nor existing). It is also unsafe to call virtual member functions through `this` in the constructor, as any derived class functions will not be considered (due to the derived class not yet being constructed, and thus its constructor not yet updating the vtable).

Also note that while in a constructor, the type of the object is the type which that constructor constructs. This holds true even if the object is declared as a derived type. For example, in the below example, `ctd_good` and `ctd_bad` are type `CtorThisBase` inside `CtorThisBase()`, and type `CtorThis` inside `CtorThis()`, even though their canonical type is `CtorThisDerived`. As the more-derived classes are constructed around the base class, the instance gradually goes through the class hierarchy until it is a fully-constructed instance of its intended type.

```
class CtorThisBase {
    short s;

  public:
    CtorThisBase() : s(516) {}
};

class CtorThis : public CtorThisBase {
    int i, j, k;

  public:
    // Good constructor.
    CtorThis() : i(s + 42), j(this->i), k(j) {}

    // Bad constructor.
    CtorThis(int ii) : i(ii), j(this->k), k(b ? 51 : -51) {
        virt_func();
    }

    virtual void virt_func() { i += 2; }
};

class CtorThisDerived : public CtorThis {
    bool b;

  public:
    CtorThisDerived()       : b(true) {}
    CtorThisDerived(int ii) : CtorThis(ii), b(false) {}

    void virt_func() override { k += (2 * i); }
};

// ...

CtorThisDerived ctd_good;
CtorThisDerived ctd_bad(3);
```

With these classes and member functions:

- In the good constructor, for `ctd_good`:
  - `CtorThisBase` is fully constructed by the time the `CtorThis` constructor is entered. Therefore, `s` is in a valid state while initialising `i`, and can thus be accessed.
  - `i` is initialised before `j(this->i)` is reached. Therefore, `i` is in a valid state while initialising `j`, and can thus be accessed.
  - `j` is initialised before `k(j)` is reached. Therefore, `j` is in a valid state while initialising `k`, and can thus be accessed.
- In the bad constructor, for `ctd_bad`:
  - `k` is initialised after `j(this->k)` is reached. Therefore, `k` is in an invalid state while initialising `j`, and accessing it causes undefined behaviour.
  - `CtorThisDerived` is not constructed until after `CtorThis` is constructed. Therefore, `b` is in an invalid state while initialising `k`, and accessing it causes undefined behaviour.
  - The object `ctd_bad` is still a `CtorThis` until it leaves `CtorThis()`, and will not be updated to use

CtorThisDerived's vtable until CtorThisDerived(). Therefore, virt_func() will call CtorThis::virt_func(), regardless of whether it is intended to call that or CtorThisDerived::virt_func().

Section 32.2: Using the this Pointer to Access Member Data

In this context, using the this pointer isn't entirely necessary, but it will make your code clearer to the reader, by indicating that a given function or variable is a member of the class. An example in this situation:

```
// Example for this pointer
#include <iostream>
#include <string>

using std::cout;
using std::endl;

class Class
{
  public:
    Class();
    ~Class();
    int getPrivateNumber () const;
  private:
    int private_number = 42;
};

Class::Class(){}
Class::~Class(){}

int Class::getPrivateNumber() const
{
    return this->private_number;
}

int main()
{
    Class class_example;
    cout << class_example.getPrivateNumber() << endl;
}
```

See it in action here.

Section 32.3: Using the this Pointer to Differentiate Between Member Data and Parameters

This is an actual useful strategy to differentiate member data from parameters... Lets take this example :

```
// Dog Class Example
#include <iostream>
#include <string>

using std::cout;
using std::endl;

/*
* @class Dog
*   @member name
*       Dog's name
*   @function bark
```

```
*       Dog Barks!
*   @function getName
*       To Get Private
*       Name Variable
*/
class Dog
{
 public:
    Dog(std::string name);
    ~Dog();
    void  bark() const;
    std::string  getName() const;
 private:
    std::string name;
};

Dog::Dog(std::string name)
{
    /*
    *  this->name is the
    *  name variable from
    *  the class dog . and
    *  name is from the
    *  parameter of the function
    */
    this->name = name;
}

Dog::~Dog(){}

void Dog::bark() const
{
  cout << "BARK" << endl;
}

std::string  Dog::getName() const
{
    return this->name;
}


int main()
{
    Dog dog("Max");
    cout << dog.getName() << endl;
    dog.bark();
}
```

You can see here in the constructor we execute the following:

```
this->name = name;
```

Here , you can see we are assinging the parameter name to the name of the private variable from the class Dog(this->name) .

To see the output of above code : http://cpp.sh/75r7

Section 32.4: this Pointer CV-Qualifiers

`this` can also be cv-qualified, the same as any other pointer. However, due to the `this` parameter not being listed

in the parameter list, special syntax is required for this; the cv-qualifiers are listed after the parameter list, but before the function's body.

```
struct ThisCVQ {
    void no_qualifier()                {} // "this" is: ThisCVQ*
    void  c_qualifier() const          {} // "this" is: const ThisCVQ*
    void  v_qualifier() volatile       {} // "this" is: volatile ThisCVQ*
    void cv_qualifier() const volatile {} // "this" is: const volatile ThisCVQ*
};
```

As `this` is a parameter, a function can be overloaded based on its `this` cv-qualifier(s).

```
struct CVOverload {
    int func()                { return    3; }
    int func() const          { return   33; }
    int func() volatile       { return  333; }
    int func() const volatile { return 3333; }
};
```

When `this` is `const` (including `const volatile`), the function is unable to write to member variables through it, whether implicitly or explicitly. The sole exception to this is `mutable` member variables, which can be written regardless of const-ness. Due to this, `const` is used to indicate that the member function doesn't change the object's logical state (the way the object appears to the outside world), even if it does modify the physical state (the way the object looks under the hood).

> Logical state is the way the object appears to outside observers. It isn't directly tied to physical state, and indeed, might not even be stored as physical state. As long as outside observers can't see any changes, the logical state is constant, even if you flip every single bit in the object.

> Physical state, also known as bitwise state, is how the object is stored in memory. This is the object's nitty-gritty, the raw 1s and 0s that make up its data. An object is only physically constant if its representation in memory never changes.

Note that C++ bases `const`ness on logical state, not physical state.

```
class DoSomethingComplexAndOrExpensive {
    mutable ResultType cached_result;
    mutable bool state_changed;

    ResultType calculate_result();
    void modify_somehow(const Param& p);

    // ...

  public:
    DoSomethingComplexAndOrExpensive(Param p) : state_changed(true) {
        modify_somehow(p);
    }

    void change_state(Param p) {
        modify_somehow(p);
        state_changed = true;
    }
```

```
    // Return some complex and/or expensive-to-calculate result.
    // As this has no reason to modify logical state, it is marked as "const".
    ResultType get_result() const;
};
ResultType DoSomethingComplexAndOrExpensive::get_result() const {
    // cached_result and state_changed can be modified, even with a const "this" pointer.
    // Even though the function doesn't modify logical state, it does modify physical state
    //  by caching the result, so it doesn't need to be recalculated every time the function
    //  is called.  This is indicated by cached_result and state_changed being mutable.

    if (state_changed) {
        cached_result = calculate_result();
        state_changed = false;
    }

    return cached_result;
}
```

Note that while you technically *could* use `const_cast` on `this` to make it non-cv-qualified, you really, ***REALLY*** shouldn't, and should use `mutable` instead. A `const_cast` is liable to invoke undefined behaviour when used on an object that actually *is* `const`, while `mutable` is designed to be safe to use. It is, however, possible that you may run into this in extremely old code.

An exception to this rule is defining non-cv-qualified accessors in terms of `const` accessors; as the object is guaranteed to not be `const` if the non-cv-qualified version is called, there's no risk of UB.

```
class CVAccessor {
    int arr[5];

  public:
    const int& get_arr_element(size_t i) const { return arr[i]; }

    int& get_arr_element(size_t i) {
        return const_cast<int&>(const_cast<const CVAccessor*>(this)->get_arr_element(i));
    }
};
```

This prevents unnecessary duplication of code.

As with regular pointers, if `this` is `volatile` (including `const volatile`), it is loaded from memory each time it is accessed, instead of being cached. This has the same effects on optimisation as declaring any other pointer `volatile` would, so care should be taken.

Note that if an instance is cv-qualified, the only member functions it is allowed to access are member functions whose `this` pointer is at least as cv-qualified as the instance itself:

- Non-cv instances can access any member functions.
- `const` instances can access `const` and `const volatile` functions.
- `volatile` instances can access `volatile` and `const volatile` functions.
- `const volatile` instances can access `const volatile` functions.

This is one of the key tenets of `const` correctness.

```
struct CVAccess {
    void    func()                {}
    void  func_c() const          {}
    void  func_v() volatile       {}
    void func_cv() const volatile {}
```

```
};

CVAccess cva;
cva.func();    // Good.
cva.func_c();  // Good.
cva.func_v();  // Good.
cva.func_cv(); // Good.

const CVAccess c_cva;
c_cva.func();    // Error.
c_cva.func_c();  // Good.
c_cva.func_v();  // Error.
c_cva.func_cv(); // Good.

volatile CVAccess v_cva;
v_cva.func();    // Error.
v_cva.func_c();  // Error.
v_cva.func_v();  // Good.
v_cva.func_cv(); // Good.

const volatile CVAccess cv_cva;
cv_cva.func();    // Error.
cv_cva.func_c();  // Error.
cv_cva.func_v();  // Error.
cv_cva.func_cv(); // Good.
```

Section 32.5: this Pointer Ref-Qualifiers

Version ≥ C++11

Similarly to `this` cv-qualifiers, we can also apply *ref-qualifiers* to `*this`. Ref-qualifiers are used to choose between normal and rvalue reference semantics, allowing the compiler to use either copy or move semantics depending on which are more appropriate, and are applied to `*this` instead of `this`.

Note that despite ref-qualifiers using reference syntax, `this` itself is still a pointer. Also note that ref-qualifiers don't actually change the type of `*this`; it's just easier to describe and understand their effects by looking at them as if they did.

```
struct RefQualifiers {
    std::string s;

    RefQualifiers(const std::string& ss = "The nameless one.") : s(ss) {}

    // Normal version.
    void func() &  { std::cout << "Accessed on normal instance "    << s << std::endl; }
    // Rvalue version.
    void func() && { std::cout << "Accessed on temporary instance " << s << std::endl; }

    const std::string& still_a_pointer() &  { return this->s; }
    const std::string& still_a_pointer() && { this->s = "Bob"; return this->s; }
};

// ...

RefQualifiers rf("Fred");
rf.func();              // Output:  Accessed on normal instance Fred
RefQualifiers{}.func(); // Output:  Accessed on temporary instance The nameless one
```

A member function cannot have overloads both with and without ref-qualifiers; the programmer has to choose

between one or the other. Thankfully, cv-qualifiers can be used in conjunction with ref-qualifiers, allowing `const` correctness rules to be followed.

```
struct RefCV {
    void func() &                {}
    void func() &&               {}
    void func() const&           {}
    void func() const&&          {}
    void func() volatile&        {}
    void func() volatile&&       {}
    void func() const volatile&  {}
    void func() const volatile&& {}
};
```

Chapter 33: Smart Pointers

Section 33.1: Unique ownership (std::unique_ptr)

Version ≥ C++11

A std::unique_ptr is a class template that manages the lifetime of a dynamically stored object. Unlike for std::shared_ptr, the dynamic object is owned by only *one instance* of a std::unique_ptr at any time,

```
// Creates a dynamic int with value of 20 owned by a unique pointer
std::unique_ptr<int> ptr = std::make_unique<int>(20);
```

(Note: std::unique_ptr is available since C++11 and std::make_unique since C++14.)

Only the variable ptr holds a pointer to a dynamically allocated int. When a unique pointer that owns an object goes out of scope, the owned object is deleted, i.e. its destructor is called if the object is of class type, and the memory for that object is released.

To use std::unique_ptr and std::make_unique with array-types, use their array specializations:

```
// Creates a unique_ptr to an int with value 59
std::unique_ptr<int> ptr = std::make_unique<int>(59);

// Creates a unique_ptr to an array of 15 ints
std::unique_ptr<int[]> ptr = std::make_unique<int[]>(15);
```

You can access the std::unique_ptr just like a raw pointer, because it overloads those operators.

You can transfer ownership of the contents of a smart pointer to another pointer by using std::move, which will cause the original smart pointer to point to nullptr.

```
// 1. std::unique_ptr
std::unique_ptr<int> ptr = std::make_unique<int>();

// Change value to 1
*ptr = 1;

// 2. std::unique_ptr (by moving 'ptr' to 'ptr2', 'ptr' doesn't own the object anymore)
std::unique_ptr<int> ptr2 = std::move(ptr);

int a = *ptr2; // 'a' is 1
int b = *ptr;  // undefined behavior! 'ptr' is 'nullptr'
               // (because of the move command above)
```

Passing unique_ptr to functions as parameter:

```
void foo(std::unique_ptr<int> ptr)
{
    // Your code goes here
}

std::unique_ptr<int> ptr = std::make_unique<int>(59);
foo(std::move(ptr))
```

Returning unique_ptr from functions. This is the preferred C++11 way of writing factory functions, as it clearly

conveys the ownership semantics of the return: the caller owns the resulting `unique_ptr` and is responsible for it.

```
std::unique_ptr<int> foo()
{
    std::unique_ptr<int> ptr = std::make_unique<int>(59);
    return ptr;
}

std::unique_ptr<int> ptr = foo();
```

Compare this to:

```
int* foo_cpp03();

int* p = foo_cpp03(); // do I own p? do I have to delete it at some point?
                      // it's not readily apparent what the answer is.
```

Version < C++14

The class template `make_unique` is provided since C++14. It's easy to add it manually to C++11 code:

```
template<typename T, typename... Args>
typename std::enable_if<!std::is_array<T>::value, std::unique_ptr<T>>::type
make_unique(Args&&... args)
{ return std::unique_ptr<T>(new T(std::forward<Args>(args)...)); }

// Use make_unique for arrays
template<typename T>
typename std::enable_if<std::is_array<T>::value, std::unique_ptr<T>>::type
make_unique(size_t n)
{ return std::unique_ptr<T>(new typename std::remove_extent<T>::type[n]()); }
```

Version ≥ C++11

Unlike the *dumb* smart pointer (`std::auto_ptr`), `unique_ptr` can also be instantiated with vector allocation (*not* `std::vector`). Earlier examples were for *scalar* allocations. For example to have a dynamically allocated integer array for 10 elements, you would specify `int[]` as the template type (and not just `int`):

```
std::unique_ptr<int[]> arr_ptr = std::make_unique<int[]>(10);
```

Which can be simplified with:

```
auto arr_ptr = std::make_unique<int[]>(10);
```

Now, you use `arr_ptr` as if it is an array:

```
arr_ptr[2] =  10; // Modify third element
```

You need not to worry about de-allocation. This template specialized version calls constructors and destructors appropriately. Using vectored version of `unique_ptr` or a `vector` itself - is a personal choice.

In versions prior to C++11, `std::auto_ptr` was available. Unlike `unique_ptr` it is allowed to copy `auto_ptr`s, upon which the source `ptr` will lose the ownership of the contained pointer and the target receives it.

Section 33.2: Sharing ownership (std::shared_ptr)

The class template `std::shared_ptr` defines a shared pointer that is able to share ownership of an object with

other shared pointers. This contrasts to `std::unique_ptr` which represents exclusive ownership.

The sharing behavior is implemented through a technique known as reference counting, where the number of shared pointers that point to the object is stored alongside it. When this count reaches zero, either through the destruction or reassignment of the last `std::shared_ptr` instance, the object is automatically destroyed.

```
// Creation: 'firstShared' is a shared pointer for a new instance of 'Foo'
std::shared_ptr<Foo> firstShared = std::make_shared<Foo>(/*args*/);
```

To create multiple smart pointers that share the same object, we need to create another `shared_ptr` that aliases the first shared pointer. Here are 2 ways of doing it:

```
std::shared_ptr<Foo> secondShared(firstShared);  // 1st way: Copy constructing
std::shared_ptr<Foo> secondShared;
secondShared = firstShared;                      // 2nd way: Assigning
```

Either of the above ways makes `secondShared` a shared pointer that shares ownership of our instance of `Foo` with `firstShared`.

The smart pointer works just like a raw pointer. This means, you can use `*` to dereference them. The regular `->` operator works as well:

```
secondShared->test(); // Calls Foo::test()
```

Finally, when the last aliased `shared_ptr` goes out of scope, the destructor of our `Foo` instance is called.

Warning: Constructing a `shared_ptr` might throw a `bad_alloc` exception when extra data for shared ownership semantics needs to be allocated. If the constructor is passed a regular pointer it assumes to own the object pointed to and calls the deleter if an exception is thrown. This means `shared_ptr<T>(new T(args))` will not leak a `T` object if allocation of `shared_ptr<T>` fails. However, it is advisable to use `make_shared<T>(args)` or `allocate_shared<T>(alloc, args)`, which enable the implementation to optimize the memory allocation.

Allocating Arrays([]) using shared_ptr

Version ≥ C++11 Version < C++17

Unfortunately, there is no direct way to allocate Arrays using `make_shared<>`.

It is possible to create arrays for `shared_ptr<>` using `new` and `std::default_delete`.

For example, to allocate an array of 10 integers, we can write the code as

```
shared_ptr<int> sh(new int[10], std::default_delete<int[]>());
```

Specifying `std::default_delete` is mandatory here to make sure that the allocated memory is correctly cleaned up using `delete[]`.

If we know the size at compile time, we can do it this way:

```
template<class Arr>
struct shared_array_maker {};
template<class T, std::size_t N>
struct shared_array_maker<T[N]> {
  std::shared_ptr<T> operator()const{
```

```
    auto r = std::make_shared<std::array<T,N>>();
    if (!r) return {};
    return {r.data(), r};
  }
};
template<class Arr>
auto make_shared_array()
-> decltype( shared_array_maker<Arr>{}() )
{ return shared_array_maker<Arr>{}(); }
```

then `make_shared_array<int[10]>` returns a `shared_ptr<int>` pointing to 10 ints all default constructed.

Version ≥ C++17

With C++17, `shared_ptr` gained special support for array types. It is no longer necessary to specify the array-deleter explicitly, and the shared pointer can be dereferenced using the `[]` array index operator:

```
std::shared_ptr<int[]> sh(new int[10]);
sh[0] = 42;
```

Shared pointers can point to a sub-object of the object it owns:

```
struct Foo { int x; };
std::shared_ptr<Foo> p1 = std::make_shared<Foo>();
std::shared_ptr<int> p2(p1, &p1->x);
```

Both `p2` and `p1` own the object of type `Foo`, but `p2` points to its `int` member `x`. This means that if `p1` goes out of scope or is reassigned, the underlying `Foo` object will still be alive, ensuring that `p2` does not dangle.

Important: A `shared_ptr` only knows about itself and all other `shared_ptr` that were created with the alias constructor. It does not know about any other pointers, including all other `shared_ptr`s created with a reference to the same `Foo` instance:

```
Foo *foo = new Foo;
std::shared_ptr<Foo> shared1(foo);
std::shared_ptr<Foo> shared2(foo); // don't do this

shared1.reset(); // this will delete foo, since shared1
                 // was the only shared_ptr that owned it

shared2->test(); // UNDEFINED BEHAVIOR: shared2's foo has been
                 // deleted already!!
```

Ownership Transfer of shared_ptr

By default, `shared_ptr` increments the reference count and doesn't transfer the ownership. However, it can be made to transfer the ownership using `std::move`:

```
shared_ptr<int> up = make_shared<int>();
// Transferring the ownership
shared_ptr<int> up2 = move(up);
// At this point, the reference count of up = 0 and the
// ownership of the pointer is solely with up2 with reference count = 1
```

Section 33.3: Sharing with temporary ownership

(std::weak_ptr)

Instances of std::weak_ptr can point to objects owned by instances of std::shared_ptr while only becoming temporary owners themselves. This means that weak pointers do not alter the object's reference count and therefore do not prevent an object's deletion if all of the object's shared pointers are reassigned or destroyed.

In the following example instances of std::weak_ptr are used so that the destruction of a tree object is not inhibited:

```
#include <memory>
#include <vector>

struct TreeNode {
    std::weak_ptr<TreeNode> parent;
    std::vector< std::shared_ptr<TreeNode> > children;
};

int main() {
    // Create a TreeNode to serve as the root/parent.
    std::shared_ptr<TreeNode> root(new TreeNode);

    // Give the parent 100 child nodes.
    for (size_t i = 0; i < 100; ++i) {
        std::shared_ptr<TreeNode> child(new TreeNode);
        root->children.push_back(child);
        child->parent = root;
    }

    // Reset the root shared pointer, destroying the root object, and
    // subsequently its child nodes.
    root.reset();
}
```

As child nodes are added to the root node's children, their std::weak_ptr member parent is set to the root node. The member parent is declared as a weak pointer as opposed to a shared pointer such that the root node's reference count is not incremented. When the root node is reset at the end of main(), the root is destroyed. Since the only remaining std::shared_ptr references to the child nodes were contained in the root's collection children, all child nodes are subsequently destroyed as well.

Due to control block implementation details, shared_ptr allocated memory may not be released until shared_ptr reference counter and weak_ptr reference counter both reach zero.

```
#include <memory>
int main()
{
    {
         std::weak_ptr<int> wk;
         {
             // std::make_shared is optimized by allocating only once
             // while std::shared_ptr<int>(new int(42)) allocates twice.
             // Drawback of std::make_shared is that control block is tied to our integer
             std::shared_ptr<int> sh = std::make_shared<int>(42);
             wk = sh;
             // sh memory should be released at this point...
         }
         // ... but wk is still alive and needs access to control block
     }
     // now memory is released (sh and wk)
```

```
}
```

Since `std::weak_ptr` does not keep its referenced object alive, direct data access through a `std::weak_ptr` is not possible. Instead it provides a `lock()` member function that attempts to retrieve a `std::shared_ptr` to the referenced object:

```
#include <cassert>
#include <memory>
int main()
{
    {
         std::weak_ptr<int> wk;
         std::shared_ptr<int> sp;
         {
             std::shared_ptr<int> sh = std::make_shared<int>(42);
             wk = sh;
             // calling lock will create a shared_ptr to the object referenced by wk
             sp = wk.lock();
             // sh will be destroyed after this point, but sp is still alive
         }
         // sp still keeps the data alive.
         // At this point we could even call lock() again
         // to retrieve another shared_ptr to the same data from wk
         assert(*sp == 42);
         assert(!wk.expired());
         // resetting sp will delete the data,
         // as it is currently the last shared_ptr with ownership
         sp.reset();
         // attempting to lock wk now will return an empty shared_ptr,
         // as the data has already been deleted
         sp = wk.lock();
         assert(!sp);
         assert(wk.expired());
     }
}
```

Section 33.4: Using custom deleters to create a wrapper to a C interface

Many C interfaces such as SDL2 have their own deletion functions. This means that you cannot use smart pointers directly:

```
std::unique_ptr<SDL_Surface> a; // won't work, UNSAFE!
```

Instead, you need to define your own deleter. The examples here use the `SDL_Surface` structure which should be freed using the `SDL_FreeSurface()` function, but they should be adaptable to many other C interfaces.

The deleter must be callable with a pointer argument, and therefore can be e.g. a simple function pointer:

```
std::unique_ptr<SDL_Surface, void(*)(SDL_Surface*)> a(pointer, SDL_FreeSurface);
```

Any other callable object will work, too, for example a class with an `operator()`:

```
struct SurfaceDeleter {
    void operator()(SDL_Surface* surf) {
        SDL_FreeSurface(surf);
    }
};
```

```
std::unique_ptr<SDL_Surface, SurfaceDeleter> a(pointer, SurfaceDeleter{}); // safe
std::unique_ptr<SDL_Surface, SurfaceDeleter> b(pointer); // equivalent to the above
                                                          // as the deleter is value-initialized
```

This not only provides you with safe, zero overhead (if you use `unique_ptr`) automatic memory management, you also get exception safety.

Note that the deleter is part of the type for `unique_ptr`, and the implementation can use the empty base optimization to avoid any change in size for empty custom deleters. So while `std::unique_ptr<SDL_Surface, SurfaceDeleter>` and `std::unique_ptr<SDL_Surface, void(*)(SDL_Surface*)>` solve the same problem in a similar way, the former type is still only the size of a pointer while the latter type has to hold *two* pointers: both the `SDL_Surface*` and the function pointer! When having free function custom deleters, it is preferable to wrap the function in an empty type.

In cases where reference counting is important, one could use a `shared_ptr` instead of an `unique_ptr`. The `shared_ptr` always stores a deleter, this erases the type of the deleter, which might be useful in APIs. The disadvantages of using `shared_ptr` over `unique_ptr` include a higher memory cost for storing the deleter and a performance cost for maintaining the reference count.

```
// deleter required at construction time and is part of the type
std::unique_ptr<SDL_Surface, void(*)(SDL_Surface*)> a(pointer, SDL_FreeSurface);

// deleter is only required at construction time, not part of the type
std::shared_ptr<SDL_Surface> b(pointer, SDL_FreeSurface);
```

Version ≥ C++17

With `template auto`, we can make it even easier to wrap our custom deleters:

```
template <auto DeleteFn>
struct FunctionDeleter {
    template <class T>
    void operator()(T* ptr) {
        DeleteFn(ptr);
    }
};

template <class T, auto DeleteFn>
using unique_ptr_deleter = std::unique_ptr<T, FunctionDeleter<DeleteFn>>;
```

With which the above example is simply:

```
unique_ptr_deleter<SDL_Surface, SDL_FreeSurface> c(pointer);
```

Here, the purpose of `auto` is to handle all free functions, whether they return `void` (e.g. `SDL_FreeSurface`) or not (e.g. `fclose`).

Section 33.5: Unique ownership without move semantics (auto_ptr)

Version < C++11

NOTE: `std::auto_ptr` has been deprecated in C++11 and will be removed in C++17. You should only use this if you are forced to use C++03 or earlier and are willing to be careful. It is recommended to move to unique_ptr in combination with `std::move` to replace `std::auto_ptr` behavior.

Before we had `std::unique_ptr`, before we had move semantics, we had `std::auto_ptr`. `std::auto_ptr` provides unique ownership but transfers ownership upon copy.

As with all smart pointers, `std::auto_ptr` automatically cleans up resources (see RAII):

```
{
    std::auto_ptr<int> p(new int(42));
    std::cout << *p;
} // p is deleted here, no memory leaked
```

but allows only one owner:

```
std::auto_ptr<X> px = ...;
std::auto_ptr<X> py = px;
  // px is now empty
```

This allows to use std::auto_ptr to keep ownership explicit and unique at the danger of losing ownership unintended:

```
void f(std::auto_ptr<X> ) {
    // assumes ownership of X
    // deletes it at end of scope
};

std::auto_ptr<X> px = ...;
f(px); // f acquires ownership of underlying X
       // px is now empty
px->foo(); // NPE!
// px.~auto_ptr() does NOT delete
```

The transfer of ownership happened in the "copy" constructor. `auto_ptr`'s copy constructor and copy assignment operator take their operands by non-`const` reference so that they could be modified. An example implementation might be:

```
template <typename T>
class auto_ptr {
    T* ptr;
public:
    auto_ptr(auto_ptr& rhs)
    : ptr(rhs.release())
    { }

    auto_ptr& operator=(auto_ptr& rhs) {
        reset(rhs.release());
        return *this;
    }

    T* release() {
        T* tmp = ptr;
        ptr = nullptr;
        return tmp;
    }

    void reset(T* tmp = nullptr) {
        if (ptr != tmp) {
            delete ptr;
            ptr = tmp;
        }
    }
```

```
    /* other functions ... */
};
```

This breaks copy semantics, which require that copying an object leaves you with two equivalent versions of it. For any copyable type, `T`, I should be able to write:

```
T a = ...;
T b(a);
assert(b == a);
```

But for `auto_ptr`, this is not the case. As a result, it is not safe to put `auto_ptr`s in containers.

Section 33.6: Casting std::shared_ptr pointers

It is not possible to directly use `static_cast`, `const_cast`, `dynamic_cast` and `reinterpret_cast` on `std::shared_ptr` to retrieve a pointer sharing ownership with the pointer being passed as argument. Instead, the functions `std::static_pointer_cast`, `std::const_pointer_cast`, `std::dynamic_pointer_cast` and `std::reinterpret_pointer_cast` should be used:

```
struct Base { virtual ~Base() noexcept {}; };
struct Derived: Base {};
auto derivedPtr(std::make_shared<Derived>());
auto basePtr(std::static_pointer_cast<Base>(derivedPtr));
auto constBasePtr(std::const_pointer_cast<Base const>(basePtr));
auto constDerivedPtr(std::dynamic_pointer_cast<Derived const>(constBasePtr));
```

Note that `std::reinterpret_pointer_cast` is not available in C++11 and C++14, as it was only proposed by N3920 and adopted into Library Fundamentals TS in February 2014. However, it can be implemented as follows:

```
template <typename To, typename From>
inline std::shared_ptr<To> reinterpret_pointer_cast(
    std::shared_ptr<From> const & ptr) noexcept
{ return std::shared_ptr<To>(ptr, reinterpret_cast<To *>(ptr.get())); }
```

Section 33.7: Writing a smart pointer: value_ptr

A `value_ptr` is a smart pointer that behaves like a value. When copied, it copies its contents. When created, it creates its contents.

```
// Like std::default_delete:
template<class T>
struct default_copier {
  // a copier must handle a null T const* in and return null:
  T* operator()(T const* tin)const {
    if (!tin) return nullptr;
    return new T(*tin);
  }
  void operator()(void* dest, T const* tin)const {
    if (!tin) return;
    return new(dest) T(*tin);
  }
};
// tag class to handle empty case:
struct empty_ptr_t {};
constexpr empty_ptr_t empty_ptr{};
// the value pointer type itself:
```

```
template<class T, class Copier=default_copier<T>, class Deleter=std::default_delete<T>,
  class Base=std::unique_ptr<T, Deleter>
>
struct value_ptr:Base, private Copier {
  using copier_type=Copier;
  // also typedefs from unique_ptr

  using Base::Base;

  value_ptr( T const& t ):
    Base( std::make_unique<T>(t) ),
    Copier()
  {}
  value_ptr( T && t ):
    Base( std::make_unique<T>(std::move(t)) ),
    Copier()
  {}
  // almost-never-empty:
      value_ptr():
    Base( std::make_unique<T>() ),
    Copier()
  {}
  value_ptr( empty_ptr_t ) {}

  value_ptr( Base b, Copier c={} ):
    Base(std::move(b)),
    Copier(std::move(c))
  {}

  Copier const& get_copier() const {
    return *this;
  }

  value_ptr clone() const {
    return {
      Base(
        get_copier()(this->get()),
        this->get_deleter()
      ),
      get_copier()
    };
  }
  value_ptr(value_ptr&&)=default;
  value_ptr& operator=(value_ptr&&)=default;

  value_ptr(value_ptr const& o):value_ptr(o.clone()) {}
  value_ptr& operator=(value_ptr const&o) {
    if (o && *this) {
      // if we are both non-null, assign contents:
      **this = *o;
    } else {
      // otherwise, assign a clone (which could itself be null):
      *this = o.clone();
    }
    return *this;
  }
  value_ptr& operator=( T const& t ) {
    if (*this) {
      **this = t;
    } else {
      *this = value_ptr(t);
    }
```

```
    return *this;
  }
  value_ptr& operator=( T && t ) {
    if (*this) {
      **this = std::move(t);
    } else {
      *this = value_ptr(std::move(t));
    }
    return *this;
  }
  T& get() { return **this; }
  T const& get() const { return **this; }
  T* get_pointer() {
    if (!*this) return nullptr;
    return std::addressof(get());
  }
  T const* get_pointer() const {
    if (!*this) return nullptr;
    return std::addressof(get());
  }
  // operator-> from unique_ptr
};
template<class T, class...Args>
value_ptr<T> make_value_ptr( Args&&... args ) {
  return {std::make_unique<T>(std::forward<Args>(args)...)};
}
```

This particular value_ptr is only empty if you construct it with `empty_ptr_t` or if you move from it. It exposes the fact it is a `unique_ptr`, so `explicit operator bool() const` works on it. `.get()` has been changed to return a reference (as it is almost never empty), and `.get_pointer()` returns a pointer instead.

This smart pointer can be useful for `pImpl` cases, where we want value-semantics but we also don't want to expose the contents of the `pImpl` outside of the implementation file.

With a non-default `Copier`, it can even handle virtual base classes that know how to produce instances of their derived and turn them into value-types.

Section 33.8: Getting a shared_ptr referring to this

`enable_shared_from_this` enables you to get a valid `shared_ptr` instance to `this`.

By deriving your class from the class template `enable_shared_from_this`, you inherit a method `shared_from_this` that returns a `shared_ptr` instance to `this`.

Note that the object must be created as a `shared_ptr` in first place:

```
#include <memory>
class A: public enable_shared_from_this<A> {
};
A* ap1 =new A();
shared_ptr<A> ap2(ap1); // First prepare a shared pointer to the object and hold it!
// Then get a shared pointer to the object from the object itself
shared_ptr<A> ap3 = ap1->shared_from_this();
int c3 =ap3.use_count(); // =2: pointing to the same object
```

Note(2) you cannot call `enable_shared_from_this` inside the constructor.

```
#include <memory> // enable_shared_from_this
```

```
class Widget : public std::enable_shared_from_this< Widget >
{
public:
    void DoSomething()
    {
        std::shared_ptr< Widget > self = shared_from_this();
        someEvent -> Register( self );
    }
private:
    ...
};

int main()
{
    ...
    auto w = std::make_shared< Widget >();
    w -> DoSomething();
    ...
}
```

If you use `shared_from_this()` on an object not owned by a `shared_ptr`, such as a local automatic object or a global object, then the behavior is undefined. Since C++17 it throws `std::bad_alloc` instead.

Using `shared_from_this()` from a constructor is equivalent to using it on an object not owned by a `shared_ptr`, because the objects is possessed by the `shared_ptr` after the constructor returns.

Chapter 34: Classes/Structures

Section 34.1: Class basics

A *class* is a user-defined type. A class is introduced with the `class`, `struct` or `union` keyword. In colloquial usage, the term "class" usually refers only to non-union classes.

A class is a collection of *class members*, which can be:

- member variables (also called "fields"),
- member functions (also called "methods"),
- member types or typedefs (e.g. "nested classes"),
- member templates (of any kind: variable, function, class or alias template)

The `class` and `struct` keywords, called *class keys*, are largely interchangeable, except that the default access specifier for members and bases is "private" for a class declared with the `class` key and "public" for a class declared with the `struct` or `union` key (cf. Access modifiers).

For example, the following code snippets are identical:

```
struct Vector
{
    int x;
    int y;
    int z;
};
// are equivalent to
class Vector
{
public:
    int x;
    int y;
    int z;
};
```

By declaring a class` a new type is added to your program, and it is possible to instantiate objects of that class by

```
Vector my_vector;
```

Members of a class are accessed using dot-syntax.

```
my_vector.x = 10;
my_vector.y = my_vector.x + 1; // my_vector.y = 11;
my_vector.z = my_vector.y - 4; // my:vector.z = 7;
```

Section 34.2: Final classes and structs

Version ≥ C++11

Deriving a class may be forbidden with `final` specifier. Let's declare a final class:

```
class A final {
};
```

Now any attempt to subclass it will cause a compilation error:

```
// Compilation error: cannot derive from final class:
class B : public A {
};
```

Final class may appear anywhere in class hierarchy:

```
class A {
};

// OK.
class B final : public A {
};

// Compilation error: cannot derive from final class B.
class C : public B {
};
```

Section 34.3: Access specifiers

There are three keywords that act as **access specifiers**. These limit the access to class members following the specifier, until another specifier changes the access level again:

Keyword	Description
`public`	Everyone has access
`protected`	Only the class itself, derived classes and friends have access
`private`	Only the class itself and friends have access

When the type is defined using the `class` keyword, the default access specifier is `private`, but if the type is defined using the `struct` keyword, the default access specifier is `public`:

```
struct MyStruct { int x; };
class MyClass { int x; };

MyStruct s;
s.x = 9; // well formed, because x is public

MyClass c;
c.x = 9; // ill-formed, because x is private
```

Access specifiers are mostly used to limit access to internal fields and methods, and force the programmer to use a specific interface, for example to force use of getters and setters instead of referencing a variable directly:

```
class MyClass {

public: /* Methods: */

    int x() const noexcept { return m_x; }
    void setX(int const x) noexcept { m_x = x; }

private: /* Fields: */

    int m_x;

};
```

Using `protected` is useful for allowing certain functionality of the type to be only accessible to the derived classes, for example, in the following code, the method `calculateValue()` is only accessible to classes deriving from the

base class `Plus2Base`, such as `FortyTwo`:

```
struct Plus2Base {
    int value() noexcept { return calculateValue() + 2; }
protected: /* Methods: */
    virtual int calculateValue() noexcept = 0;
};
struct FortyTwo: Plus2Base {
protected: /* Methods: */
    int calculateValue() noexcept final override { return 40; }
};
```

Note that the `friend` keyword can be used to add access exceptions to functions or types for accessing protected and private members.

The `public`, `protected`, and `private` keywords can also be used to grant or limit access to base class subobjects. See the Inheritance example.

Section 34.4: Inheritance

Classes/structs can have inheritance relations.

If a class/struct `B` inherits from a class/struct `A`, this means that `B` has as a parent `A`. We say that `B` is a derived class/struct from `A`, and `A` is the base class/struct.

```
struct A
{
public:
    int p1;
protected:
    int p2;
private:
    int p3;
};

//Make B inherit publicly (default) from A
struct B : A
{
};
```

There are 3 forms of inheritance for a class/struct:

- `public`
- `private`
- `protected`

Note that the default inheritance is the same as the default visibility of members: `public` if you use the `struct` keyword, and `private` for the `class` keyword.

It's even possible to have a `class` derive from a `struct` (or vice versa). In this case, the default inheritance is controlled by the child, so a `struct` that derives from a `class` will default to public inheritance, and a `class` that derives from a `struct` will have private inheritance by default.

`public` inheritance:

```
struct B : public A // or just `struct B : A`
{
```

```
    void foo()
    {
        p1 = 0; //well formed, p1 is public in B
        p2 = 0; //well formed, p2 is protected in B
        p3 = 0; //ill formed, p3 is private in A
    }
};

B b;
b.p1 = 1; //well formed, p1 is public
b.p2 = 1; //ill formed, p2 is protected
b.p3 = 1; //ill formed, p3 is inaccessible
```

`private` inheritance:

```
struct B : private A
{
    void foo()
    {
        p1 = 0; //well formed, p1 is private in B
        p2 = 0; //well formed, p2 is private in B
        p3 = 0; //ill formed, p3 is private in A
    }
};

B b;
b.p1 = 1; //ill formed, p1 is private
b.p2 = 1; //ill formed, p2 is private
b.p3 = 1; //ill formed, p3 is inaccessible
```

`protected` inheritance:

```
struct B : protected A
{
    void foo()
    {
        p1 = 0; //well formed, p1 is protected in B
        p2 = 0; //well formed, p2 is protected in B
        p3 = 0; //ill formed, p3 is private in A
    }
};

B b;
b.p1 = 1; //ill formed, p1 is protected
b.p2 = 1; //ill formed, p2 is protected
b.p3 = 1; //ill formed, p3 is inaccessible
```

Note that although `protected` inheritance is allowed, the actual use of it is rare. One instance of how `protected` inheritance is used in application is in partial base class specialization (usually referred to as "controlled polymorphism").

When OOP was relatively new, (public) inheritance was frequently said to model an "IS-A" relationship. That is, public inheritance is correct only if an instance of the derived class *is also an* instance of the base class.

This was later refined into the Liskov Substitution Principle: public inheritance should only be used when/if an instance of the derived class can be substituted for an instance of the base class under any possible circumstance (and still make sense).

Private inheritance is typically said to embody a completely different relationship: "is implemented in terms of"

(sometimes called a "HAS-A" relationship). For example, a `Stack` class could inherit privately from a `Vector` class. Private inheritance bears a much greater similarity to aggregation than to public inheritance.

Protected inheritance is almost never used, and there's no general agreement on what sort of relationship it embodies.

Section 34.5: Friendship

The `friend` keyword is used to give other classes and functions access to private and protected members of the class, even through they are defined outside the class`s scope.

```
class Animal{
private:
    double weight;
    double height;
public:
    friend void printWeight(Animal animal);
    friend class AnimalPrinter;
    // A common use for a friend function is to overload the operator<< for streaming.
    friend std::ostream& operator<<(std::ostream& os, Animal animal);
};

void printWeight(Animal animal)
{
    std::cout << animal.weight << "\n";
}

class AnimalPrinter
{
public:
    void print(const Animal& animal)
    {
        // Because of the `friend class AnimalPrinter;" declaration, we are
        // allowed to access private members here.
        std::cout << animal.weight << ", " << animal.height << std::endl;
    }
}

std::ostream& operator<<(std::ostream& os, Animal animal)
{
    os << "Animal height: " << animal.height << "\n";
    return os;
}

int main() {
    Animal animal = {10, 5};
    printWeight(animal);

    AnimalPrinter aPrinter;
    aPrinter.print(animal);

    std::cout << animal;
}
```

```
10
10, 5
Animal height: 5
```

Section 34.6: Virtual Inheritance

When using inheritance, you can specify the `virtual` keyword:

```
struct A{};
struct B: public virtual A{};
```

When class `B` has virtual base `A` it means that `A` **will reside in most derived class** of inheritance tree, and thus that most derived class is also responsible for initializing that virtual base:

```
struct A
{
    int member;
    A(int param)
    {
        member = param;
    }
};

struct B: virtual A
{
    B(): A(5){}
};

struct C: B
{
    C(): /*A(88)*/ {}
};

void f()
{
    C object; //error since C is not initializing it's indirect virtual base `A`
}
```

If we un-comment `/*A(88)*/` we won't get any error since `C` is now initializing it's indirect virtual base `A`.

Also note that when we're creating variable `object`, most derived class is `C`, so `C` is responsible for creating(calling constructor of) `A` and thus value of `A::member` is `88`, not `5` (as it would be if we were creating object of type `B`).

It is useful when solving the diamond problem.:

```
  A                                       A   A
 / \                                      |   |
B   C                                     B   C
 \ /                                       \ /
  D                                         D
virtual inheritance                    normal inheritance
```

`B` and `C` both inherit from `A`, and `D` inherits from `B` and `C`, so **there are 2 instances of `A` in `D`!** This results in ambiguity when you're accessing member of `A` through `D`, as the compiler has no way of knowing from which class do you want to access that member (the one which `B` inherits, or the one that is inherited by`C`?).

Virtual inheritance solves this problem: Since virtual base resides only in most derived object, there will be only one instance of `A` in `D`.

```
struct A
{
    void foo() {}
```

```
};

struct B : public /*virtual*/ A {};
struct C : public /*virtual*/ A {};

struct D : public B, public C
{
    void bar()
    {
        foo(); //Error, which foo? B::foo() or C::foo()? - Ambiguous
    }
};
```

Removing the comments resolves the ambiguity.

Section 34.7: Private inheritance: restricting base class interface

Private inheritance is useful when it is required to restrict the public interface of the class:

```
class A {
public:
    int move();
    int turn();
};

class B : private A {
public:
    using A::turn;
};

B b;
b.move();  // compile error
b.turn();  // OK
```

This approach efficiently prevents an access to the A public methods by casting to the A pointer or reference:

```
B b;
A& a = static_cast<A&>(b); // compile error
```

In the case of public inheritance such casting will provide access to all the A public methods despite on alternative ways to prevent this in derived B, like hiding:

```
class B : public A {
private:
    int move();
};
```

or private using:

```
class B : public A {
private:
    using A::move;
};
```

then for both cases it is possible:

```
B b;
```

```
A& a = static_cast<A&>(b); // OK for public inheritance
a.move(); // OK
```

Section 34.8: Accessing class members

To access member variables and member functions of an object of a class, the `.` operator is used:

```
struct SomeStruct {
  int a;
  int b;
  void foo() {}
};

SomeStruct var;
// Accessing member variable a in var.
std::cout << var.a << std::endl;
// Assigning member variable b in var.
var.b = 1;
// Calling a member function.
var.foo();
```

When accessing the members of a class via a pointer, the `->` operator is commonly used. Alternatively, the instance can be dereferenced and the `.` operator used, although this is less common:

```
struct SomeStruct {
  int a;
  int b;
  void foo() {}
};

SomeStruct var;
SomeStruct *p = &var;
// Accessing member variable a in var via pointer.
std::cout << p->a << std::endl;
std::cout << (*p).a << std::endl;
// Assigning member variable b in var via pointer.
p->b = 1;
(*p).b = 1;
// Calling a member function via a pointer.
p->foo();
(*p).foo();
```

When accessing static class members, the `::` operator is used, but on the name of the class instead of an instance of it. Alternatively, the static member can be accessed from an instance or a pointer to an instance using the `.` or `->` operator, respectively, with the same syntax as accessing non-static members.

```
struct SomeStruct {
  int a;
  int b;
  void foo() {}

  static int c;
  static void bar() {}
};
int SomeStruct::c;

SomeStruct var;
SomeStruct* p = &var;
// Assigning static member variable c in struct SomeStruct.
```

```
SomeStruct::c = 5;
// Accessing static member variable c in struct SomeStruct, through var and p.
var.a = var.c;
var.b = p->c;
// Calling a static member function.
SomeStruct::bar();
var.bar();
p->bar();
```

Background

The `->` operator is needed because the member access operator `.` has precedence over the dereferencing operator `*`.

One would expect that `*p.a` would dereference `p` (resulting in a reference to the object `p` is pointing to) and then accessing its member `a`. But in fact, it tries to access the member `a` of `p` and then dereference it. I.e. `*p.a` is equivalent to `*(p.a)`. In the example above, this would result in a compiler error because of two facts: First, `p` is a pointer and does not have a member `a`. Second, `a` is an integer and, thus, can't be dereferenced.

The uncommonly used solution to this problem would be to explicitly control the precedence: `(*p).a`

Instead, the `->` operator is almost always used. It is a short-hand for first dereferencing the pointer and then accessing it. I.e. `(*p).a` is exactly the same as `p->a`.

The `::` operator is the scope operator, used in the same manner as accessing a member of a namespace. This is because a static class member is considered to be in that class' scope, but isn't considered a member of instances of that class. The use of normal `.` and `->` is also allowed for static members, despite them not being instance members, for historical reasons; this is of use for writing generic code in templates, as the caller doesn't need to be concerned with whether a given member function is static or non-static.

Section 34.9: Member Types and Aliases

A `class` or `struct` can also define member type aliases, which are type aliases contained within, and treated as members of, the class itself.

```
struct IHaveATypedef {
    typedef int MyTypedef;
};

struct IHaveATemplateTypedef {
    template<typename T>
    using MyTemplateTypedef = std::vector<T>;
};
```

Like static members, these typedefs are accessed using the scope operator, `::`.

```
IHaveATypedef::MyTypedef i = 5; // i is an int.

IHaveATemplateTypedef::MyTemplateTypedef<int> v; // v is a std::vector<int>.
```

As with normal type aliases, each member type alias is allowed to refer to any type defined or aliased before, but not after, its definition. Likewise, a typedef outside the class definition can refer to any accessible typedefs within the class definition, provided it comes after the class definition.

```
template<typename T>
struct Helper {
```

```
    T get() const { return static_cast<T>(42); }
};

struct IHaveTypedefs {
//    typedef MyTypedef NonLinearTypedef; // Error if uncommented.
    typedef int MyTypedef;
    typedef Helper<MyTypedef> MyTypedefHelper;
};

IHaveTypedefs::MyTypedef        i; // x_i is an int.
IHaveTypedefs::MyTypedefHelper hi; // x_hi is a Helper<int>.

typedef IHaveTypedefs::MyTypedef TypedefBeFree;
TypedefBeFree ii;                  // ii is an int.
```

Member type aliases can be declared with any access level, and will respect the appropriate access modifier.

```
class TypedefAccessLevels {
    typedef int PrvInt;

  protected:
    typedef int ProInt;

  public:
    typedef int PubInt;
};

TypedefAccessLevels::PrvInt prv_i; // Error: TypedefAccessLevels::PrvInt is private.
TypedefAccessLevels::ProInt pro_i; // Error: TypedefAccessLevels::ProInt is protected.
TypedefAccessLevels::PubInt pub_i; // Good.

class Derived : public TypedefAccessLevels {
    PrvInt prv_i; // Error: TypedefAccessLevels::PrvInt is private.
    ProInt pro_i; // Good.
    PubInt pub_i; // Good.
};
```

This can be used to provide a level of abstraction, allowing a class' designer to change its internal workings without breaking code that relies on it.

```
class Something {
    friend class SomeComplexType;

    short s;
    // ...

  public:
    typedef SomeComplexType MyHelper;

    MyHelper get_helper() const { return MyHelper(8, s, 19.5, "shoe", false); }

    // ...
};

// ...

Something s;
Something::MyHelper hlp = s.get_helper();
```

In this situation, if the helper class is changed from `SomeComplexType` to some other type, only the `typedef` and the

friend declaration would need to be modified; as long as the helper class provides the same functionality, any code that uses it as Something::MyHelper instead of specifying it by name will usually still work without any modifications. In this manner, we minimise the amount of code that needs to be modified when the underlying implementation is changed, such that the type name only needs to be changed in one location.

This can also be combined with decltype, if one so desires.

```
class SomethingElse {
    AnotherComplexType<bool, int, SomeThirdClass> helper;

  public:
    typedef decltype(helper) MyHelper;

  private:
    InternalVariable<MyHelper> ivh;

    // ...

  public:
    MyHelper& get_helper() const { return helper; }

    // ...
};
```

In this situation, changing the implementation of SomethingElse::helper will automatically change the typedef for us, due to decltype. This minimises the number of modifications necessary when we want to change helper, which minimises the risk of human error.

As with everything, however, this can be taken too far. If the typename is only used once or twice internally and zero times externally, for example, there's no need to provide an alias for it. If it's used hundreds or thousands of times throughout a project, or if it has a long enough name, then it can be useful to provide it as a typedef instead of always using it in absolute terms. One must balance forwards compatibility and convenience with the amount of unnecessary noise created.

This can also be used with template classes, to provide access to the template parameters from outside the class.

```
template<typename T>
class SomeClass {
    // ...

  public:
    typedef T MyParam;
    MyParam getParam() { return static_cast<T>(42); }
};

template<typename T>
typename T::MyParam some_func(T& t) {
    return t.getParam();
}

SomeClass<int> si;
int i = some_func(si);
```

This is commonly used with containers, which will usually provide their element type, and other helper types, as member type aliases. Most of the containers in the C++ standard library, for example, provide the following 12 helper types, along with any other special types they might need.

```
template<typename T>
```

```
class SomeContainer {
    // ...

  public:
    // Let's provide the same helper types as most standard containers.
    typedef T                                      value_type;
    typedef std::allocator<value_type>             allocator_type;
    typedef value_type&                            reference;
    typedef const value_type&                      const_reference;
    typedef value_type*                            pointer;
    typedef const value_type*                      const_pointer;
    typedef MyIterator<value_type>                 iterator;
    typedef MyConstIterator<value_type>            const_iterator;
    typedef std::reverse_iterator<iterator>        reverse_iterator;
    typedef std::reverse_iterator<const_iterator> const_reverse_iterator;
    typedef size_t                                 size_type;
    typedef ptrdiff_t                              difference_type;
};
```

Prior to C++11, it was also commonly used to provide a "template `typedef`" of sorts, as the feature wasn't yet available; these have become a bit less common with the introduction of alias templates, but are still useful in some situations (and are combined with alias templates in other situations, which can be very useful for obtaining individual components of a complex type such as a function pointer). They commonly use the name `type` for their type alias.

```
template<typename T>
struct TemplateTypedef {
    typedef T type;
}

TemplateTypedef<int>::type i; // i is an int.
```

This was often used with types with multiple template parameters, to provide an alias that defines one or more of the parameters.

```
template<typename T, size_t SZ, size_t D>
class Array { /* ... */ };

template<typename T, size_t SZ>
struct OneDArray {
    typedef Array<T, SZ, 1> type;
};

template<typename T, size_t SZ>
struct TwoDArray {
    typedef Array<T, SZ, 2> type;
};

template<typename T>
struct MonoDisplayLine {
    typedef Array<T, 80, 1> type;
};

OneDArray<int, 3>::type     arr1i; // arr1i is an Array<int, 3, 1>.
TwoDArray<short, 5>::type   arr2s; // arr2s is an Array<short, 5, 2>.
MonoDisplayLine<char>::type arr3c; // arr3c is an Array<char, 80, 1>.
```

Section 34.10: Nested Classes/Structures

A `class` or `struct` can also contain another `class`/`struct` definition inside itself, which is called a "nested class"; in this situation, the containing class is referred to as the "enclosing class". The nested class definition is considered to be a member of the enclosing class, but is otherwise separate.

```
struct Outer {
    struct Inner { };
};
```

From outside of the enclosing class, nested classes are accessed using the scope operator. From inside the enclosing class, however, nested classes can be used without qualifiers:

```
struct Outer {
    struct Inner { };

    Inner in;
};

// ...

Outer o;
Outer::Inner i = o.in;
```

As with a non-nested `class`/`struct`, member functions and static variables can be defined either within a nested class, or in the enclosing namespace. However, they cannot be defined within the enclosing class, due to it being considered to be a different class than the nested class.

```
// Bad.
struct Outer {
    struct Inner {
        void do_something();
    };

    void Inner::do_something() {}
};


// Good.
struct Outer {
    struct Inner {
        void do_something();
    };

};

void Outer::Inner::do_something() {}
```

As with non-nested classes, nested classes can be forward declared and defined later, provided they are defined before being used directly.

```
class Outer {
    class Inner1;
    class Inner2;

    class Inner1 {};

    Inner1 in1;
```

```
    Inner2* in2p;

  public:
    Outer();
    ~Outer();
};

class Outer::Inner2 {};

Outer::Outer() : in1(Inner1()), in2p(new Inner2) {}
Outer::~Outer() {
    if (in2p) { delete in2p; }
}
```

Version < C++11

Prior to C++11, nested classes only had access to type names, `static` members, and enumerators from the enclosing class; all other members defined in the enclosing class were off-limits.

Version ≥ C++11

As of C++11, nested classes, and members thereof, are treated as if they were `friend`s of the enclosing class, and can access all of its members, according to the usual access rules; if members of the nested class require the ability to evaluate one or more non-static members of the enclosing class, they must therefore be passed an instance:

```
class Outer {
    struct Inner {
        int get_sizeof_x() {
            return sizeof(x); // Legal (C++11): x is unevaluated, so no instance is required.
        }

        int get_x() {
            return x; // Illegal: Can't access non-static member without an instance.
        }

        int get_x(Outer& o) {
            return o.x; // Legal (C++11): As a member of Outer, Inner can access private members.
        }
    };

    int x;
};
```

Conversely, the enclosing class is *not* treated as a friend of the nested class, and thus cannot access its private members without explicitly being granted permission.

```
class Outer {
    class Inner {
        // friend class Outer;

        int x;
    };

    Inner in;

  public:
    int get_x() {
        return in.x; // Error: int Outer::Inner::x is private.
        // Uncomment "friend" line above to fix.
    }
```

```
};
```

Friends of a nested class are not automatically considered friends of the enclosing class; if they need to be friends of the enclosing class as well, this must be declared separately. Conversely, as the enclosing class is not automatically considered a friend of the nested class, neither will friends of the enclosing class be considered friends of the nested class.

```
class Outer {
    friend void barge_out(Outer& out, Inner& in);

    class Inner {
        friend void barge_in(Outer& out, Inner& in);

        int i;
    };

    int o;
};

void barge_in(Outer& out, Outer::Inner& in) {
    int i = in.i;  // Good.
    int o = out.o; // Error: int Outer::o is private.
}

void barge_out(Outer& out, Outer::Inner& in) {
    int i = in.i;  // Error: int Outer::Inner::i is private.
    int o = out.o; // Good.
}
```

As with all other class members, nested classes can only be named from outside the class if they have public access. However, you are allowed to access them regardless of access modifier, as long as you don't explicitly name them.

```
class Outer {
    struct Inner {
        void func() { std::cout << "I have no private taboo.\n"; }
    };

  public:
    static Inner make_Inner() { return Inner(); }
};

// ...

Outer::Inner oi; // Error: Outer::Inner is private.

auto oi = Outer::make_Inner(); // Good.
oi.func();                     // Good.
Outer::make_Inner().func();    // Good.
```

You can also create a type alias for a nested class. If a type alias is contained in the enclosing class, the nested type and the type alias can have different access modifiers. If the type alias is outside the enclosing class, it requires that either the nested class, or a `typedef` thereof, be public.

```
class Outer {
    class Inner_ {};

  public:
    typedef Inner_ Inner;
```

```
};

typedef Outer::Inner  ImOut; // Good.
typedef Outer::Inner_ ImBad; // Error.

// ...

Outer::Inner  oi; // Good.
Outer::Inner_ oi; // Error.
ImOut         oi; // Good.
```

As with other classes, nested classes can both derive from or be derived from by other classes.

```
struct Base {};

struct Outer {
    struct Inner : Base {};
};

struct Derived : Outer::Inner {};
```

This can be useful in situations where the enclosing class is derived from by another class, by allowing the programmer to update the nested class as necessary. This can be combined with a typedef to provide a consistent name for each enclosing class' nested class:

```
class BaseOuter {
    struct BaseInner_ {
        virtual void do_something() {}
        virtual void do_something_else();
    } b_in;

  public:
    typedef BaseInner_ Inner;

    virtual ~BaseOuter() = default;

    virtual Inner& getInner() { return b_in; }
};

void BaseOuter::BaseInner_::do_something_else() {}

// ---

class DerivedOuter : public BaseOuter {
    // Note the use of the qualified typedef; BaseOuter::BaseInner_ is private.
    struct DerivedInner_ : BaseOuter::Inner {
        void do_something() override {}
        void do_something_else() override;
    } d_in;

  public:
    typedef DerivedInner_ Inner;

    BaseOuter::Inner& getInner() override { return d_in; }
};

void DerivedOuter::DerivedInner_::do_something_else() {}

// ...
```

```
// Calls BaseOuter::BaseInner_::do_something();
BaseOuter* b = new BaseOuter;
BaseOuter::Inner& bin = b->getInner();
bin.do_something();
b->getInner().do_something();

// Calls DerivedOuter::DerivedInner_::do_something();
BaseOuter* d = new DerivedOuter;
BaseOuter::Inner& din = d->getInner();
din.do_something();
d->getInner().do_something();
```

In the above case, both `BaseOuter` and `DerivedOuter` supply the member type `Inner`, as `BaseInner_` and `DerivedInner_`, respectively. This allows nested types to be derived without breaking the enclosing class' interface, and allows the nested type to be used polymorphically.

Section 34.11: Unnamed struct/class

Unnamed `struct` is allowed (type has no name)

```
void foo()
{
    struct /* No name */ {
        float x;
        float y;
    } point;

    point.x = 42;
}
```

or

```
struct Circle
{
    struct /* No name */ {
        float x;
        float y;
    } center; // but a member name
    float radius;
};
```

and later

```
Circle circle;
circle.center.x = 42.f;
```

but NOT *anonymous* `struct` (unnamed type and unnamed object)

```
struct InvalidCircle
{
    struct /* No name */ {
        float centerX;
        float centerY;
    }; // No member either.
    float radius;
};
```

Note: Some compilers allow *anonymous* `struct` as *extension*.

Version ≥ C++11

- *lamdba* can be seen as a special *unnamed* `struct`.
- `decltype` allows to retrieve the type of *unnamed* `struct`:

```
decltype(circle.point) otherPoint;
```

- *unnamed* `struct` instance can be parameter of template method:

```
void print_square_coordinates()
{
    const struct {float x; float y;} points[] = {
        {-1, -1}, {-1, 1}, {1, -1}, {1, 1}
    };

    // for range relies on `template <class T, std::size_t N> std::begin(T (&)[N])`
    for (const auto& point : points) {
        std::cout << "{" << point.x << ", " << point.y << "}\n";
    }

    decltype(points[0]) topRightCorner{1, 1};
    auto it = std::find(points, points + 4, topRightCorner);
    std::cout << "top right corner is the "
              << 1 + std::distance(points, it) << "th\n";
}
```

Section 34.12: Static class members

A class is also allowed to have `static` members, which can be either variables or functions. These are considered to be in the class' scope, but aren't treated as normal members; they have static storage duration (they exist from the start of the program to the end), aren't tied to a particular instance of the class, and only one copy exists for the entire class.

```
class Example {
    static int num_instances;      // Static data member (static member variable).
    int i;                         // Non-static member variable.

  public:
    static std::string static_str; // Static data member (static member variable).
    static int static_func();      // Static member function.

    // Non-static member functions can modify static member variables.
    Example() { ++num_instances; }
    void set_str(const std::string& str);
};

int         Example::num_instances;
std::string Example::static_str = "Hello.";

// ...

Example one, two, three;
// Each Example has its own "i", such that:
//  (&one.i != &two.i)
//  (&one.i != &three.i)
//  (&two.i != &three.i).
// All three Examples share "num_instances", such that:
```

```
//   (&one.num_instances == &two.num_instances)
//   (&one.num_instances == &three.num_instances)
//   (&two.num_instances == &three.num_instances)
```

Static member variables are not considered to be defined inside the class, only declared, and thus have their definition outside the class definition; the programmer is allowed, but not required, to initialise static variables in their definition. When defining the member variables, the keyword `static` is omitted.

```
class Example {
    static int num_instances;               // Declaration.

  public:
    static std::string static_str;          // Declaration.

    // ...
};

int         Example::num_instances;         // Definition.  Zero-initialised.
std::string Example::static_str = "Hello."; // Definition.
```

Due to this, static variables can be incomplete types (apart from `void`), as long as they're later defined as a complete type.

```
struct ForwardDeclared;

class ExIncomplete {
    static ForwardDeclared fd;
    static ExIncomplete    i_contain_myself;
    static int             an_array[];
};

struct ForwardDeclared {};

ForwardDeclared ExIncomplete::fd;
ExIncomplete    ExIncomplete::i_contain_myself;
int             ExIncomplete::an_array[5];
```

Static member functions can be defined inside or outside the class definition, as with normal member functions. As with static member variables, the keyword `static` is omitted when defining static member functions outside the class definition.

```
// For Example above, either...
class Example {
    // ...

  public:
    static int static_func() { return num_instances; }

    // ...

    void set_str(const std::string& str) { static_str = str; }
};

// Or...

class Example { /* ... */ };

int  Example::static_func() { return num_instances; }
```

```
void Example::set_str(const std::string& str) { static_str = str; }
```

If a static member variable is declared `const` but not `volatile`, and is of an integral or enumeration type, it can be initialised at declaration, inside the class definition.

```
enum E { VAL = 5 };

struct ExConst {
    const static int ci = 5;              // Good.
    static const E ce = VAL;              // Good.
    const static double cd = 5;           // Error.
    static const volatile int cvi = 5;    // Error.

    const static double good_cd;
    static const volatile int good_cvi;
};

const double ExConst::good_cd = 5;        // Good.
const volatile int ExConst::good_cvi = 5; // Good.
```

Version ≥ C++11

As of C++11, static member variables of `LiteralType` types (types that can be constructed at compile time, according to `constexpr` rules) can also be declared as `constexpr`; if so, they must be initialised within the class definition.

```
struct ExConstexpr {
    constexpr static int ci = 5;                      // Good.
    static constexpr double cd = 5;                   // Good.
    constexpr static int carr[] = { 1, 1, 2 };        // Good.
    static constexpr ConstexprConstructibleClass c{}; // Good.
    constexpr static int bad_ci;                      // Error.
};

constexpr int ExConstexpr::bad_ci = 5;                // Still an error.
```

If a `const` or `constexpr` static member variable is *odr-used* (informally, if it has its address taken or is assigned to a reference), then it must still have a separate definition, outside the class definition. This definition is not allowed to contain an initialiser.

```
struct ExODR {
    static const int odr_used = 5;
};

// const int ExODR::odr_used;

const int* odr_user = & ExODR::odr_used; // Error; uncomment above line to resolve.
```

As static members aren't tied to a given instance, they can be accessed using the scope operator, `::`.

```
std::string str = Example::static_str;
```

They can also be accessed as if they were normal, non-static members. This is of historical significance, but is used less commonly than the scope operator to prevent confusion over whether a member is static or non-static.

```
Example ex;
std::string rts = ex.static_str;
```

Class members are able to access static members without qualifying their scope, as with non-static class members.

```
class ExTwo {
    static int num_instances;
    int my_num;

  public:
    ExTwo() : my_num(num_instances++) {}

    static int get_total_instances() { return num_instances; }
    int get_instance_number() const { return my_num; }
};

int ExTwo::num_instances;
```

They cannot be `mutable`, nor would they need to be; as they aren't tied to any given instance, whether an instance is or isn't const doesn't affect static members.

```
struct ExDontNeedMutable {
    int immuta;
    mutable int muta;

    static int i;

    ExDontNeedMutable() : immuta(-5), muta(-5) {}
};
int ExDontNeedMutable::i;

// ...

const ExDontNeedMutable dnm;
dnm.immuta = 5; // Error: Can't modify read-only object.
dnm.muta = 5;   // Good.  Mutable fields of const objects can be written.
dnm.i = 5;      // Good.  Static members can be written regardless of an instance's const-ness.
```

Static members respect access modifiers, just like non-static members.

```
class ExAccess {
    static int prv_int;

  protected:
    static int pro_int;

  public:
    static int pub_int;
};

int ExAccess::prv_int;
int ExAccess::pro_int;
int ExAccess::pub_int;

// ...

int x1 = ExAccess::prv_int; // Error: int ExAccess::prv_int is private.
int x2 = ExAccess::pro_int; // Error: int ExAccess::pro_int is protected.
int x3 = ExAccess::pub_int; // Good.
```

As they aren't tied to a given instance, static member functions have no `this` pointer; due to this, they can't access non-static member variables unless passed an instance.

```
class ExInstanceRequired {
    int i;

  public:
    ExInstanceRequired() : i(0) {}

    static void bad_mutate() { ++i *= 5; }                         // Error.
    static void good_mutate(ExInstanceRequired& e) { ++e.i *= 5; } // Good.
};
```

Due to not having a `this` pointer, their addresses can't be stored in pointers-to-member-functions, and are instead stored in normal pointers-to-functions.

```
struct ExPointer {
           void nsfunc() {}
    static void  sfunc() {}
};

typedef void (ExPointer::* mem_f_ptr)();
typedef void (*f_ptr)();

mem_f_ptr p_sf = &ExPointer::sfunc; // Error.
    f_ptr p_sf = &ExPointer::sfunc; // Good.
```

Due to not having a `this` pointer, they also cannot be `const` or `volatile`, nor can they have ref-qualifiers. They also cannot be virtual.

```
struct ExCVQualifiersAndVirtual {
    static void   func()                {} // Good.
    static void  cfunc() const          {} // Error.
    static void  vfunc() volatile       {} // Error.
    static void cvfunc() const volatile {} // Error.
    static void  rfunc() &              {} // Error.
    static void rvfunc() &&             {} // Error.

    virtual static void vsfunc()        {} // Error.
    static virtual void svfunc()        {} // Error.
};
```

As they aren't tied to a given instance, static member variables are effectively treated as special global variables; they're created when the program starts, and destroyed when it exits, regardless of whether any instances of the class actually exist. Only a single copy of each static member variable exists (unless the variable is declared `thread_local` (C++11 or later), in which case there's one copy per thread).

Static member variables have the same linkage as the class, whether the class has external or internal linkage. Local classes and unnamed classes aren't allowed to have static members.

Section 34.13: Multiple Inheritance

Aside from single inheritance:

```
class A {};
class B : public A {};
```

You can also have multiple inheritance:

```
class A {};
```

```
class B {};
class C : public A, public B {};
```

C will now have inherit from A and B at the same time.

Note: this can lead to ambiguity if the same names are used in multiple inherited `class`s or `struct`s. Be careful!

Ambiguity in Multiple Inheritance

Multiple inheritance may be helpful in certain cases but, sometimes odd sort of problem encounters while using multiple inheritance.

For example: Two base classes have functions with same name which is not overridden in derived class and if you write code to access that function using object of derived class, compiler shows error because, it cannot determine which function to call. Here is a code for this type of ambiguity in multiple inheritance.

```
class base1
{
  public:
     void funtion( )
     { //code for base1 function }
};
class base2
{
    void function( )
     { // code for base2 function }
};

class derived : public base1, public base2
{

};

int main()
{
    derived obj;

  // Error because compiler can't figure out which function to call
  //either function( ) of base1 or base2 .
    obj.function( )
}
```

But, this problem can be solved using scope resolution function to specify which function to class either base1 or base2:

```
int main()
{
    obj.base1::function( );  // Function of class base1 is called.
    obj.base2::function( );  // Function of class base2 is called.
}
```

Section 34.14: Non-static member functions

A class can have non-static member functions, which operate on individual instances of the class.

```
class CL {
  public:
    void member_function() {}
```

```
};
```

These functions are called on an instance of the class, like so:

```
CL instance;
instance.member_function();
```

They can be defined either inside or outside the class definition; if defined outside, they are specified as being in the class' scope.

```
struct ST {
    void  defined_inside() {}
    void defined_outside();
};
void ST::defined_outside() {}
```

They can be CV-qualified and/or ref-qualified, affecting how they see the instance they're called upon; the function will see the instance as having the specified cv-qualifier(s), if any. Which version is called will be based on the instance's cv-qualifiers. If there is no version with the same cv-qualifiers as the instance, then a more-cv-qualified version will be called if available.

```
struct CVQualifiers {
    void func()                   {} // 1: Instance is non-cv-qualified.
    void func() const             {} // 2: Instance is const.

    void cv_only() const volatile {}
};

CVQualifiers       non_cv_instance;
const CVQualifiers      c_instance;

non_cv_instance.func(); // Calls #1.
c_instance.func();      // Calls #2.

non_cv_instance.cv_only(); // Calls const volatile version.
c_instance.cv_only();      // Calls const volatile version.
```

Version ≥ C++11

Member function ref-qualifiers indicate whether or not the function is intended to be called on rvalue instances, and use the same syntax as function cv-qualifiers.

```
struct RefQualifiers {
    void func() &  {} // 1: Called on normal instances.
    void func() && {} // 2: Called on rvalue (temporary) instances.
};

RefQualifiers rf;
rf.func();              // Calls #1.
RefQualifiers{}.func(); // Calls #2.
```

CV-qualifiers and ref-qualifiers can also be combined if necessary.

```
struct BothCVAndRef {
    void func() const& {} // Called on normal instances.  Sees instance as const.
    void func() &&     {} // Called on temporary instances.
};
```

They can also be virtual; this is fundamental to polymorphism, and allows a child class(es) to provide the same interface as the parent class, while supplying their own functionality.

```
struct Base {
    virtual void func() {}
};
struct Derived {
    virtual void func() {}
};

Base* bp = new Base;
Base* dp = new Derived;
bp.func(); // Calls Base::func().
dp.func(); // Calls Derived::func().
```

For more information, see here.

Chapter 35: Function Overloading

See also separate topic on Overload Resolution

Section 35.1: What is Function Overloading?

Function overloading is having multiple functions declared in the same scope with the exact same name exist in the same place (known as *scope*) differing only in their *signature*, meaning the arguments they accept.

Suppose you are writing a series of functions for generalized printing capabilities, beginning with `std::string`:

```
void print(const std::string &str)
{
    std::cout << "This is a string: " << str << std::endl;
}
```

This works fine, but suppose you want a function that also accepts an `int` and prints that too. You could write:

```
void print_int(int num)
{
    std::cout << "This is an int:  " << num << std::endl;
}
```

But because the two functions accept different parameters, you can simply write:

```
void print(int num)
{
    std::cout << "This is an int: " << num << std::endl;
}
```

Now you have 2 functions, both named `print`, but with different signatures. One accepts `std::string`, the other one an `int`. Now you can call them without worrying about different names:

```
print("Hello world!"); //prints "This is a string: Hello world!"
print(1337);           //prints "This is an int: 1337"
```

Instead of:

```
print("Hello world!");
print_int(1337);
```

When you have overloaded functions, the compiler infers which of the functions to call from the parameters you provide it. Care must be taken when writing function overloads. For example, with implicit type conversions:

```
void print(int num)
{
    std::cout << "This is an int: " << num << std::endl;
}
void print(double num)
{
    std::cout << "This is a double: " << num << std::endl;
}
```

Now it's not immediately clear which overload of `print` is called when you write:

```
print(5);
```

And you might need to give your compiler some clues, like:

```
print(static_cast<double>(5));
print(static_cast<int>(5));
print(5.0);
```

Some care also needs to be taken when writing overloads that accept optional parameters:

```
// WRONG CODE
void print(int num1, int num2 = 0)    //num2 defaults to 0 if not included
{
    std::cout << "These are ints: << num1 << " and " << num2 << std::endl;
}
void print(int num)
{
    std::cout << "This is an int: " << num << std::endl;
}
```

Because there's no way for the compiler to tell if a call like `print(17)` is meant for the first or second function because of the optional second parameter, this will fail to compile.

Section 35.2: Return Type in Function Overloading

Note that you cannot overload a function based on its return type. For example:

```
// WRONG CODE
std::string getValue()
{
  return "hello";
}

int getValue()
{
  return 0;
}

int x = getValue();
```

This will cause a compilation error as the compiler will not be able to work out which version of `getValue` to call, even though the return type is assigned to an `int`.

Section 35.3: Member Function cv-qualifier Overloading

Functions within a class can be overloaded for when they are accessed through a cv-qualified reference to that class; this is most commonly used to overload for `const`, but can be used to overload for `volatile` and `const volatile`, too. This is because all non-static member functions take `this` as a hidden parameter, which the cv-qualifiers are applied to. This is most commonly used to overload for `const`, but can also be used for `volatile` and `const volatile`.

This is necessary because a member function can only be called if it is at least as cv-qualified as the instance it's called on. While a non-`const` instance can call both `const` and non-`const` members, a `const` instance can only call `const` members. This allows a function to have different behaviour depending on the calling instance's cv-qualifiers, and allows the programmer to disallow functions for an undesired cv-qualifier(s) by not providing a version with that qualifier(s).

A class with some basic `print` method could be `const` overloaded like so:

```
#include <iostream>

class Integer
{
    public:
        Integer(int i_): i{i_}{}

        void print()
        {
            std::cout << "int: " << i << std::endl;
        }

        void print() const
        {
            std::cout << "const int: " << i << std::endl;
        }

    protected:
        int i;
};

int main()
{
    Integer i{5};
    const Integer &ic = i;

    i.print(); // prints "int: 5"
    ic.print(); // prints "const int: 5"
}
```

This is a key tenet of `const` correctness: By marking member functions as `const`, they are allowed to be called on `const` instances, which in turn allows functions to take instances as `const` pointers/references if they don't need to modify them. This allows code to specify whether it modifies state by taking unmodified parameters as `const` and modified parameters without cv-qualifiers, making code both safer and more readable.

```
class ConstCorrect
{
  public:
    void good_func() const
    {
        std::cout << "I care not whether the instance is const." << std::endl;
    }

    void bad_func()
    {
        std::cout << "I can only be called on non-const, non-volatile instances." << std::endl;
    }
};

void i_change_no_state(const ConstCorrect& cc)
{
    std::cout << "I can take either a const or a non-const ConstCorrect." << std::endl;
    cc.good_func(); // Good.  Can be called from const or non-const instance.
    cc.bad_func();  // Error.  Can only be called from non-const instance.
}

void const_incorrect_func(ConstCorrect& cc)
{
```

```
    cc.good_func(); // Good.  Can be called from const or non-const instance.
    cc.bad_func();  // Good.  Can only be called from non-const instance.
}
```

A common usage of this is declaring accessors as `const`, and mutators as non-`const`.

No class members can be modified within a `const` member function. If there is some member that you really need to modify, such as locking a `std::mutex`, you can declare it as `mutable`:

```
class Integer
{
    public:
        Integer(int i_): i{i_}{}

        int get() const
        {
            std::lock_guard<std::mutex> lock{mut};
            return i;
        }

        void set(int i_)
        {
            std::lock_guard<std::mutex> lock{mut};
            i = i_;
        }

    protected:
        int i;
        mutable std::mutex mut;
};
```

Chapter 36: Operator Overloading

In C++, it is possible to define operators such as `+` and `->` for user-defined types. For example, the **<string>** header defines a `+` operator to concatenate strings. This is done by defining an *operator function* using the `operator` keyword.

Section 36.1: Arithmetic operators

You can overload all basic arithmetic operators:

- `+` and `+=`
- `-` and `-=`
- `*` and `*=`
- `/` and `/=`
- `&` and `&=`
- `|` and `|=`
- `^` and `^=`
- `>>` and `>>=`
- `<<` and `<<=`

Overloading for all operators is the same. *Scroll down for explanation*

Overloading outside of `class`/`struct`:

```
//operator+ should be implemented in terms of operator+=
T operator+(T lhs, const T& rhs)
{
    lhs += rhs;
    return lhs;
}

T& operator+=(T& lhs, const T& rhs)
{
    //Perform addition
    return lhs;
}
```

Overloading inside of `class`/`struct`:

```
//operator+ should be implemented in terms of operator+=
T operator+(const T& rhs)
{
    *this += rhs;
    return *this;
}

T& operator+=(const T& rhs)
{
    //Perform addition
    return *this;
}
```

Note: `operator+` should return by non-const value, as returning a reference wouldn't make sense (it returns a *new* object) nor would returning a `const` value (you should generally not return by `const`). The first argument is passed

by value, why? Because

1. You can't modify the original object (`Object foobar = foo + bar;` shouldn't modify `foo` after all, it wouldn't make sense)
2. You can't make it `const`, because you will have to be able to modify the object (because `operator+` is implemented in terms of `operator+=`, which modifies the object)

Passing by `const&` would be an option, but then you will have to make a temporary copy of the passed object. By passing by value, the compiler does it for you.

`operator+=` returns a reference to the itself, because it is then possible to chain them (don't use the same variable though, that would be undefined behavior due to sequence points).

The first argument is a reference (we want to modify it), but not `const`, because then you wouldn't be able to modify it. The second argument should not be modified, and so for performance reason is passed by `const&` (passing by const reference is faster than by value).

Section 36.2: Array subscript operator

You can even overload the array subscript operator `[]`.

You should **always** (99.98% of the time) implement 2 versions, a `const` and a not-`const` version, because if the object is `const`, it should not be able to modify the object returned by `[]`.

The arguments are passed by `const&` instead of by value because passing by reference is faster than by value, and `const` so that the operator doesn't change the index accidentally.

The operators return by reference, because by design you can modify the object `[]` return, i.e:

```
std::vector<int> v{ 1 };
v[0] = 2; //Changes value of 1 to 2
          //wouldn't be possible if not returned by reference
```

You can **only** overload inside a `class`/`struct`:

```
//I is the index type, normally an int
T& operator[](const I& index)
{
    //Do something
    //return something
}

//I is the index type, normally an int
const T& operator[](const I& index) const
{
    //Do something
    //return something
}
```

Multiple subscript operators, `[][]...`, can be achieved via proxy objects. The following example of a simple row-major matrix class demonstrates this:

```
template<class T>
```

```
class matrix {
    // class enabling [][] overload to access matrix elements
    template <class C>
    class proxy_row_vector {
        using reference = decltype(std::declval<C>()[0]);
        using const_reference = decltype(std::declval<C const>()[0]);
    public:
        proxy_row_vector(C& _vec, std::size_t _r_ind, std::size_t _cols)
            : vec(_vec), row_index(_r_ind), cols(_cols) {}
        const_reference operator[](std::size_t _col_index) const {
            return vec[row_index*cols + _col_index];
        }
        reference operator[](std::size_t _col_index) {
            return vec[row_index*cols + _col_index];
        }
    private:
        C& vec;
        std::size_t row_index; // row index to access
        std::size_t cols; // number of columns in matrix
    };

    using const_proxy = proxy_row_vector<const std::vector<T>>;
    using proxy = proxy_row_vector<std::vector<T>>;
public:
    matrix() : mtx(), rows(0), cols(0) {}
    matrix(std::size_t _rows, std::size_t _cols)
        : mtx(_rows*_cols), rows(_rows), cols(_cols) {}

    // call operator[] followed by another [] call to access matrix elements
    const_proxy operator[](std::size_t _row_index) const {
        return const_proxy(mtx, _row_index, cols);
    }

    proxy operator[](std::size_t _row_index) {
        return proxy(mtx, _row_index, cols);
    }
private:
    std::vector<T> mtx;
    std::size_t rows;
    std::size_t cols;
};
```

Section 36.3: Conversion operators

You can overload type operators, so that your type can be implicitly converted into the specified type.

The conversion operator **must** be defined in a `class`/`struct`:

```
operator T() const { /* return something */ }
```

Note: the operator is `const` to allow `const` objects to be converted.

Example:

```
struct Text
{
    std::string text;

    // Now Text can be implicitly converted into a const char*
    /*explicit*/ operator const char*() const { return text.data(); }
```

```
    // ^^^^^^^
    // to disable implicit conversion
};

Text t;
t.text = "Hello world!";

//Ok
const char* copyoftext = t;
```

Section 36.4: Complex Numbers Revisited

The code below implements a very simple complex number type for which the underlying field is automatically promoted, following the language's type promotion rules, under application of the four basic operators (+, -, *, and /) with a member of a different field (be it another `complex<T>` or some scalar type).

This is intended to be a holistic example covering operator overloading alongside basic use of templates.

```
#include <type_traits>

namespace not_std{

using std::decay_t;

//-----------------------------------------------------------------
// complex< value_t >
//-----------------------------------------------------------------

template<typename value_t>
struct complex
{
    value_t x;
    value_t y;

    complex &operator += (const value_t &x)
    {
        this->x += x;
        return *this;
    }
    complex &operator += (const complex &other)
    {
        this->x += other.x;
        this->y += other.y;
        return *this;
    }

    complex &operator -= (const value_t &x)
    {
        this->x -= x;
        return *this;
    }
    complex &operator -= (const complex &other)
    {
        this->x -= other.x;
        this->y -= other.y;
        return *this;
    }

    complex &operator *= (const value_t &s)
    {
```

```
        this->x *= s;
        this->y *= s;
        return *this;
    }
    complex &operator *= (const complex &other)
    {
        (*this) = (*this) * other;
        return *this;
    }

    complex &operator /= (const value_t &s)
    {
        this->x /= s;
        this->y /= s;
        return *this;
    }
    complex &operator /= (const complex &other)
    {
        (*this) = (*this) / other;
        return *this;
    }

    complex(const value_t &x, const value_t &y)
    : x{x}
    , y{y}
    {}

    template<typename other_value_t>
    explicit complex(const complex<other_value_t> &other)
    : x{static_cast<const value_t &>(other.x)}
    , y{static_cast<const value_t &>(other.y)}
    {}

    complex &operator = (const complex &) = default;
    complex &operator = (complex &&) = default;
    complex(const complex &) = default;
    complex(complex &&) = default;
    complex() = default;
};

// Absolute value squared
template<typename value_t>
value_t absqr(const complex<value_t> &z)
{ return z.x*z.x + z.y*z.y; }

//----------------------------------------------------------------
// operator - (negation)
//----------------------------------------------------------------

template<typename value_t>
complex<value_t> operator - (const complex<value_t> &z)
{ return {-z.x, -z.y}; }

//----------------------------------------------------------------
// operator +
//----------------------------------------------------------------

template<typename left_t,typename right_t>
auto operator + (const complex<left_t> &a, const complex<right_t> &b)
-> complex<decay_t<decltype(a.x + b.x)>>
{ return{a.x + b.x, a.y + b.y}; }
```

```cpp
template<typename left_t,typename right_t>
auto operator + (const left_t &a, const complex<right_t> &b)
-> complex<decay_t<decltype(a + b.x)>>
{ return{a + b.x, b.y}; }

template<typename left_t,typename right_t>
auto operator + (const complex<left_t> &a, const right_t &b)
-> complex<decay_t<decltype(a.x + b)>>
{ return{a.x + b, a.y}; }

//------------------------------------------------------------------
// operator -
//------------------------------------------------------------------

template<typename left_t,typename right_t>
auto operator - (const complex<left_t> &a, const complex<right_t> &b)
-> complex<decay_t<decltype(a.x - b.x)>>
{ return{a.x - b.x, a.y - b.y}; }

template<typename left_t,typename right_t>
auto operator - (const left_t &a, const complex<right_t> &b)
-> complex<decay_t<decltype(a - b.x)>>
{ return{a - b.x, - b.y}; }

template<typename left_t,typename right_t>
auto operator - (const complex<left_t> &a, const right_t &b)
-> complex<decay_t<decltype(a.x - b)>>
{ return{a.x - b, a.y}; }

//------------------------------------------------------------------
// operator *
//------------------------------------------------------------------

template<typename left_t, typename right_t>
auto operator * (const complex<left_t> &a, const complex<right_t> &b)
-> complex<decay_t<decltype(a.x * b.x)>>
{
    return {
        a.x*b.x - a.y*b.y,
        a.x*b.y + a.y*b.x
        };
}

template<typename left_t, typename right_t>
auto operator * (const left_t &a, const complex<right_t> &b)
-> complex<decay_t<decltype(a * b.x)>>
{ return {a * b.x, a * b.y}; }

template<typename left_t, typename right_t>
auto operator * (const complex<left_t> &a, const right_t &b)
-> complex<decay_t<decltype(a.x * b)>>
{ return {a.x * b, a.y * b}; }

//------------------------------------------------------------------
// operator /
//------------------------------------------------------------------

template<typename left_t, typename right_t>
auto operator / (const complex<left_t> &a, const complex<right_t> &b)
-> complex<decay_t<decltype(a.x / b.x)>>
{
    const auto r = absqr(b);
```

```
    return {
        ( a.x*b.x + a.y*b.y) / r,
        (-a.x*b.y + a.y*b.x) / r
        };
}

template<typename left_t, typename right_t>
auto operator / (const left_t &a, const complex<right_t> &b)
-> complex<decay_t<decltype(a / b.x)>>
{
    const auto s = a/absqr(b);
    return {
         b.x * s,
        -b.y * s
        };
}

template<typename left_t, typename right_t>
auto operator / (const complex<left_t> &a, const right_t &b)
-> complex<decay_t<decltype(a.x / b)>>
{ return {a.x / b, a.y / b}; }

}// namespace not_std


int main(int argc, char **argv)
{
    using namespace not_std;

    complex<float> fz{4.0f, 1.0f};

    // makes a complex<double>
    auto dz = fz * 1.0;

    // still a complex<double>
    auto idz = 1.0f/dz;

    // also a complex<double>
    auto one = dz * idz;

    // a complex<double> again
    auto one_again = fz * idz;

    // Operator tests, just to make sure everything compiles.

    complex<float> a{1.0f, -2.0f};
    complex<double> b{3.0, -4.0};

    // All of these are complex<double>
    auto c0 = a + b;
    auto c1 = a - b;
    auto c2 = a * b;
    auto c3 = a / b;

    // All of these are complex<float>
    auto d0 = a + 1;
    auto d1 = 1 + a;
    auto d2 = a - 1;
    auto d3 = 1 - a;
    auto d4 = a * 1;
    auto d5 = 1 * a;
    auto d6 = a / 1;
```

```
    auto d7 = 1 / a;

    // All of these are complex<double>
    auto e0 = b + 1;
    auto e1 = 1 + b;
    auto e2 = b - 1;
    auto e3 = 1 - b;
    auto e4 = b * 1;
    auto e5 = 1 * b;
    auto e6 = b / 1;
    auto e7 = 1 / b;

    return 0;
}
```

Section 36.5: Named operators

You can extend C++ with named operators that are "quoted" by standard C++ operators.

First we start with a dozen-line library:

```
namespace named_operator {
  template<class D>struct make_operator{constexpr make_operator(){}};

  template<class T, char, class O> struct half_apply { T&& lhs; };

  template<class Lhs, class Op>
  half_apply<Lhs, '*', Op> operator*( Lhs&& lhs, make_operator<Op> ) {
    return {std::forward<Lhs>(lhs)};
  }

  template<class Lhs, class Op, class Rhs>
  auto operator*( half_apply<Lhs, '*', Op>&& lhs, Rhs&& rhs )
  -> decltype( named_invoke( std::forward<Lhs>(lhs.lhs), Op{}, std::forward<Rhs>(rhs) ) )
  {
    return named_invoke( std::forward<Lhs>(lhs.lhs), Op{}, std::forward<Rhs>(rhs) );
  }
}
```

this doesn't do anything yet.

First, appending vectors

```
namespace my_ns {
  struct append_t : named_operator::make_operator<append_t> {};
  constexpr append_t append{};

  template<class T, class A0, class A1>
  std::vector<T, A0> named_invoke( std::vector<T, A0> lhs, append_t, std::vector<T, A1> const& rhs
) {
      lhs.insert( lhs.end(), rhs.begin(), rhs.end() );
      return std::move(lhs);
  }
}
using my_ns::append;

std::vector<int> a {1,2,3};
std::vector<int> b {4,5,6};
```

```
auto c = a *append* b;
```

The core here is that we define an `append` object of type `append_t:named_operator::make_operator<append_t>`.

We then overload named_invoke(lhs, append_t, rhs) for the types we want on the right and left.

The library overloads `lhs*append_t`, returning a temporary `half_apply` object. It also overloads `half_apply*rhs` to call `named_invoke( lhs, append_t, rhs )`.

We simply have to create the proper `append_t` token and do an ADL-friendly `named_invoke` of the proper signature, and everything hooks up and works.

For a more complex example, suppose you want to have element-wise multiplication of elements of a std::array:

```
template<class=void, std::size_t...Is>
auto indexer( std::index_sequence<Is...> ) {
  return [](auto&& f) {
    return f( std::integral_constant<std::size_t, Is>{}... );
  };
}
template<std::size_t N>
auto indexer() { return indexer( std::make_index_sequence<N>{} ); }

namespace my_ns {
  struct e_times_t : named_operator::make_operator<e_times_t> {};
  constexpr e_times_t e_times{};

  template<class L, class R, std::size_t N,
    class Out=std::decay_t<decltype( std::declval<L const&>()*std::declval<R const&>() )>
  >
  std::array<Out, N> named_invoke( std::array<L, N> const& lhs, e_times_t, std::array<R, N> const&
rhs ) {
    using result_type = std::array<Out, N>;
    auto index_over_N = indexer<N>();
    return index_over_N([&](auto...is)->result_type {
      return {{
        (lhs[is] * rhs[is])...
      }};
    });
  }
}
```

live example.

This element-wise array code can be extended to work on tuples or pairs or C-style arrays, or even variable length containers if you decide what to do if the lengths don't match.

You could also an element-wise operator type and get `lhs *element_wise<'+'>* rhs`.

Writing a `*dot*` and `*cross*` product operators are also obvious uses.

The use of `*` can be extended to support other delimiters, like `+`. The delimeter precidence determines the precidence of the named operator, which may be important when translating physics equations over to C++ with minimal use of extra `()`s.

With a slight change in the library above, we can support `->*then*` operators and extend `std::function` prior to the standard being updated, or write monadic `->*bind*`. It could also have a stateful named operator, where we carefully pass the `Op` down to the final invoke function, permitting:

```
named_operator<'*'> append = [](auto lhs, auto&& rhs) {
  using std::begin; using std::end;
  lhs.insert( end(lhs), begin(rhs), end(rhs) );
  return std::move(lhs);
};
```

generating a named container-appending operator in C++17.

Section 36.6: Unary operators

You can overload the 2 unary operators:

- `++foo` and `foo++`
- `--foo` and `foo--`

Overloading is the same for both types (`++` and `--`). *Scroll down for explanation*

Overloading outside of `class`/`struct`:

```
//Prefix operator ++foo
T& operator++(T& lhs)
{
    //Perform addition
    return lhs;
}

//Postfix operator foo++ (int argument is used to separate pre- and postfix)
//Should be implemented in terms of ++foo (prefix operator)
T operator++(T& lhs, int)
{
    T t(lhs);
    ++lhs;
    return t;
}
```

Overloading inside of `class`/`struct`:

```
//Prefix operator ++foo
T& operator++()
{
    //Perform addition
    return *this;
}

//Postfix operator foo++ (int argument is used to separate pre- and postfix)
//Should be implemented in terms of ++foo (prefix operator)
T operator++(int)
{
    T t(*this);
    ++(*this);
    return t;
}
```

Note: The prefix operator returns a reference to itself, so that you can continue operations on it. The first argument is a reference, as the prefix operator changes the object, that's also the reason why it isn't `const` (you wouldn't be able to modify it otherwise).

The postfix operator returns by value a temporary (the previous value), and so it cannot be a reference, as it would be a reference to a temporary, which would be garbage value at the end of the function, because the temporary variable goes out of scope). It also cannot be `const`, because you should be able to modify it directly.

The first argument is a non-`const` reference to the "calling" object, because if it were `const`, you wouldn't be able to modify it, and if it weren't a reference, you wouldn't change the original value.

It is because of the copying needed in postfix operator overloads that it's better to make it a habit to use prefix ++ instead of postfix ++ in `for` loops. From the `for` loop perspective, they're usually functionally equivalent, but there might be a slight performance advantage to using prefix ++, especially with "fat" classes with a lot of members to copy. Example of using prefix ++ in a for loop:

```
for (list<string>::const_iterator it = tokens.begin();
     it != tokens.end();
     ++it) { // Don't use it++
    ...
}
```

Section 36.7: Comparison operators

You can overload all comparison operators:

- `==` and `!=`
- `>` and `<`
- `>=` and `<=`

The recommended way to overload all those operators is by implementing only 2 operators (`==` and `<`) and then using those to define the rest. *Scroll down for explanation*

Overloading outside of `class`/`struct`:

```
//Only implement those 2
bool operator==(const T& lhs, const T& rhs) { /* Compare */ }
bool operator<(const T& lhs, const T& rhs) { /* Compare */ }

//Now you can define the rest
bool operator!=(const T& lhs, const T& rhs) { return !(lhs == rhs); }
bool operator>(const T& lhs, const T& rhs) { return rhs < lhs; }
bool operator<=(const T& lhs, const T& rhs) { return !(lhs > rhs); }
bool operator>=(const T& lhs, const T& rhs) { return !(lhs < rhs); }
```

Overloading inside of `class`/`struct`:

```
//Note that the functions are const, because if they are not const, you wouldn't be able
//to call them if the object is const

//Only implement those 2
bool operator==(const T& rhs) const { /* Compare */ }
bool operator<(const T& rhs) const { /* Compare */ }

//Now you can define the rest
bool operator!=(const T& rhs) const { return !(*this == rhs); }
bool operator>(const T& rhs) const { return rhs < *this; }
bool operator<=(const T& rhs) const { return !(*this > rhs); }
bool operator>=(const T& rhs) const { return !(*this < rhs); }
```

The operators obviously return a `bool`, indicating `true` or `false` for the corresponding operation.

All of the operators take their arguments by `const&`, because the only thing that does operators do is compare, so they shouldn't modify the objects. Passing by `&` (reference) is faster than by value, and to make sure that the operators don't modify it, it is a `const`-reference.

Note that the operators inside the `class`/`struct` are defined as `const`, the reason for this is that without the functions being `const`, comparing `const` objects would not be possible, as the compiler doesn't know that the operators don't modify anything.

Section 36.8: Assignment operator

The assignment operator is one of the most important operators because it allows you to change the status of a variable.

If you do not overload the assignment operator for your `class`/`struct`, it is automatically generated by the compiler: the automatically-generated assignment operator performs a "memberwise assignment", ie by invoking assignment operators on all members, so that one object is copied to the other, a member at time. The assignment operator should be overloaded when the simple memberwise assignment is not suitable for your `class`/`struct`, for example if you need to perform a **deep copy** of an object.

Overloading the assignment operator `=` is easy, but you should follow some simple steps.

1. **Test for self-assignment.** This check is important for two reasons:
   - a self-assignment is a needless copy, so it does not make sense to perform it;
   - the next step will not work in the case of a self-assignment.
2. **Clean the old data.** The old data must be replaced with new ones. Now, you can understand the second reason of the previous step: if the content of the object was destroyed, a self-assignment will fail to perform the copy.
3. **Copy all members.** If you overload the assignment operator for your `class` or your `struct`, it is not automatically generated by the compiler, so you will need to take charge of copying all members from the other object.
4. **Return `*this`.** The operator returns by itself by reference, because it allows chaining (i.e. `int b = (a = 6) + 4; //b == 10`).

```
//T is some type
T& operator=(const T& other)
{
    //Do something (like copying values)
    return *this;
}
```

Note: `other` is passed by `const&`, because the object being assigned should not be changed, and passing by reference is faster than by value, and to make sure than `operator=` doesn't modify it accidentally, it is `const`.

The assignment operator can **only** to be overloaded in the `class`/`struct`, because the left value of `=` is **always** the `class`/`struct` itself. Defining it as a free function doesn't have this guarantee, and is disallowed because of that.

When you declare it in the `class`/`struct`, the left value is implicitly the `class`/`struct` itself, so there is no problem with that.

Section 36.9: Function call operator

You can overload the function call operator `()`:

Overloading must be done inside of a `class`/`struct`:

```
//R -> Return type
//Types -> any different type
R operator()(Type name, Type2 name2, ...)
{
    //Do something
    //return something
}

//Use it like this (R is return type, a and b are variables)
R foo = object(a, b, ...);
```

For example:

```
struct Sum
{
    int operator()(int a, int b)
    {
        return a + b;
    }
};

//Create instance of struct
Sum sum;
int result = sum(1, 1); //result == 2
```

Section 36.10: Bitwise NOT operator

Overloading the bitwise NOT (~) is fairly simple. *Scroll down for explanation*

Overloading outside of `class`/`struct`:

```
T operator~(T lhs)
{
    //Do operation
    return lhs;
}
```

Overloading inside of `class`/`struct`:

```
T operator~()
{
    T t(*this);
    //Do operation
    return t;
}
```

Note: `operator~` returns by value, because it has to return a new value (the modified value), and not a reference to the value (it would be a reference to the temporary object, which would have garbage value in it as soon as the operator is done). Not `const` either because the calling code should be able to modify it afterwards (i.e. `int a = ~a + 1;` should be possible).

Inside the `class`/`struct` you have to make a temporary object, because you can't modify `this`, as it would modify the original object, which shouldn't be the case.

Section 36.11: Bit shift operators for I/O

The operators `<<` and `>>` are commonly used as "write" and "read" operators:

- `std::ostream` overloads `<<` to write variables to the underlying stream (example: `std::cout`)
- `std::istream` overloads `>>` to read from the underlying stream to a variable (example: `std::cin`)

The way they do this is similar if you wanted to overload them "normally" outside of the `class`/`struct`, except that specifying the arguments are not of the same type:

- Return type is the stream you want to overload from (for example, `std::ostream`) passed by reference, to allow chaining (Chaining: `std::cout << a << b;`). Example: `std::ostream&`
- `lhs` would be the same as the return type
- `rhs` is the type you want to allow overloading from (i.e. `T`), passed by `const&` instead of value for performance reason (`rhs` shouldn't be changed anyway). Example: `const Vector&`.

Example:

```
//Overload std::ostream operator<< to allow output from Vector's
std::ostream& operator<<(std::ostream& lhs, const Vector& rhs)
{
    lhs << "x: " << rhs.x << " y: " << rhs.y << " z: " << rhs.z << '\n';
    return lhs;
}

Vector v = { 1, 2, 3};

//Now you can do
std::cout << v;
```

Chapter 37: Function Template Overloading

Section 37.1: What is a valid function template overloading?

A function template can be overloaded under the rules for non-template function overloading (same name, but different parameter types) and in addition to that, the overloading is valid if

- The return type is different, or
- The template parameter list is different, except for the naming of parameters and the presence of default arguments (they are not part of the signature)

For a normal function, comparing two parameter types is is easy for the compiler, since it has all informat. But a type within a template may not be determined yet. Therefore, the rule for when two parameter types are equal is approximative here and says that the non-depependend types and values need to match and the spelling of dependent types and expressions needs to be the same (more precisely, they need to conform to the so-called ODR-rules), except that template parameters may be renamed. However, if under such different spellings, two values within the types are deemed different, but will always instantiate to the same values, the overloading is invalid, but no diagnostic is required from the compiler.

```
template<typename T>
void f(T*) { }

template<typename T>
void f(T) { }
```

This is a valid overload, as "T" and "T*" are different spellings. But the following is invalid, with no diagnostic required

```
template<typename T>
void f(T (*x)[sizeof(T) + sizeof(T)]) { }

template<typename T>
void f(T (*x)[2 * sizeof(T)]) { }
```

Chapter 38: Virtual Member Functions

Section 38.1: Final virtual functions

C++11 introduced `final` specifier which forbids method overriding if appeared in method signature:

```
class Base {
public:
    virtual void foo() {
        std::cout << "Base::Foo\n";
    }
};

class Derived1 : public Base {
public:
    // Overriding Base::foo
    void foo() final {
        std::cout << "Derived1::Foo\n";
    }
};

class Derived2 : public Derived1 {
public:
    // Compilation error: cannot override final method
    virtual void foo() {
        std::cout << "Derived2::Foo\n";
    }
};
```

The specifier `final` can only be used with `virtual' member function and can't be applied to non-virtual member functions

Like `final`, there is also an specifier caller 'override' which prevent overriding of `virtual` functions in the derived class.

The specifiers `override` and `final` may be combined together to have desired effect:

```
class Derived1 : public Base {
public:
    void foo() final override {
        std::cout << "Derived1::Foo\n";
    }
};
```

Section 38.2: Using override with virtual in C++11 and later

The specifier `override` has a special meaning in C++11 onwards, if appended at the end of function signature. This signifies that a function is

- Overriding the function present in base class &
- The Base class function is `virtual`

There is no `run time` significance of this specifier as is mainly meant as an indication for compilers

The example below will demonstrate the change in behaviour with our without using override.

Without `override`:

```
#include <iostream>

struct X {
    virtual void f() { std::cout << "X::f()\n"; }
};

struct Y : X {
    // Y::f() will not override X::f() because it has a different signature,
    // but the compiler will accept the code (and silently ignore Y::f()).
    virtual void f(int a) { std::cout << a << "\n"; }
};
```

With `override`:

```
#include <iostream>

struct X {
    virtual void f() { std::cout << "X::f()\n"; }
};

struct Y : X {
    // The compiler will alert you to the fact that Y::f() does not
    // actually override anything.
    virtual void f(int a) override { std::cout << a << "\n"; }
};
```

Note that `override` is not a keyword, but a special identifier which only may appear in function signatures. In all other contexts `override` still may be used as an identifier:

```
void foo() {
    int override = 1; // OK.
    int virtual = 2;  // Compilation error: keywords can't be used as identifiers.
}
```

Section 38.3: Virtual vs non-virtual member functions

With virtual member functions:

```
#include <iostream>

struct X {
    virtual void f() { std::cout << "X::f()\n"; }
};

struct Y : X {
    // Specifying virtual again here is optional
    // because it can be inferred from X::f().
    virtual void f() { std::cout << "Y::f()\n"; }
};

void call(X& a) {
    a.f();
}

int main() {
    X x;
    Y y;
    call(x); // outputs "X::f()"
    call(y); // outputs "Y::f()"
```

```
}
```

Without virtual member functions:

```
#include <iostream>

struct X {
   void f() { std::cout << "X::f()\n"; }
};

struct Y : X {
   void f() { std::cout << "Y::f()\n"; }
};

void call(X& a) {
    a.f();
}

int main() {
    X x;
    Y y;
    call(x); // outputs "X::f()"
    call(y); // outputs "X::f()"
}
```

Section 38.4: Behaviour of virtual functions in constructors and destructors

The behaviour of virtual functions in constructors and destructors is often confusing when first encountered.

```
#include <iostream>
using namespace std;

class base {
public:
    base() { f("base constructor"); }
    ~base() { f("base destructor"); }

    virtual const char* v() { return "base::v()"; }

    void f(const char* caller) {
        cout << "When called from " << caller << ", "  << v() << " gets called.\n";
    }
};

class derived : public base {
public:
    derived() { f("derived constructor"); }
    ~derived() { f("derived destructor"); }

    const char* v() override { return "derived::v()"; }

};

int main() {
     derived d;
}
```

Output:

> When called from base constructor, base::v() gets called.
> When called from derived constructor, derived::v() gets called.
> When called from derived destructor, derived::v() gets called.
> When called from base destructor, base::v() gets called.

The reasoning behind this is that the derived class may define additional members which are not yet initialized (in the constructor case) or already destroyed (in the destructor case), and calling its member functions would be unsafe. Therefore during construction and destruction of C++ objects, the *dynamic* type of `*this` is considered to be the constructor's or destructor's class and not a more-derived class.

Example:

```
#include <iostream>
#include <memory>

using namespace std;
class base {
public:
    base()
    {
        std::cout << "foo is " << foo() << std::endl;
    }
    virtual int foo() { return 42; }
};

class derived : public base {
    unique_ptr<int> ptr_;
public:
    derived(int i) : ptr_(new int(i*i)) { }
    // The following cannot be called before derived::derived due to how C++ behaves,
    // if it was possible... Kaboom!
    int foo() override   { return *ptr_; }
};

int main() {
    derived d(4);
}
```

Section 38.5: Pure virtual functions

We can also specify that a `virtual` function is *pure virtual* (abstract), by appending `= 0` to the declaration. Classes with one or more pure virtual functions are considered to be abstract, and cannot be instantiated; only derived classes which define, or inherit definitions for, all pure virtual functions can be instantiated.

```
struct Abstract {
    virtual void f() = 0;
};

struct Concrete {
    void f() override {}
};

Abstract a; // Error.
Concrete c; // Good.
```

Even if a function is specified as pure virtual, it can be given a default implementation. Despite this, the function will still be considered abstract, and derived classes will have to define it before they can be instantiated. In this case,

the derived class' version of the function is even allowed to call the base class' version.

```
struct DefaultAbstract {
    virtual void f() = 0;
};
void DefaultAbstract::f() {}

struct WhyWouldWeDoThis : DefaultAbstract {
    void f() override { DefaultAbstract::f(); }
};
```

There are a couple of reasons why we might want to do this:

- If we want to create a class that can't itself be instantiated, but doesn't prevent its derived classes from being instantiated, we can declare the destructor as pure virtual. Being the destructor, it must be defined anyways, if we want to be able to deallocate the instance. And as the destructor is most likely already virtual to prevent memory leaks during polymorphic use, we won't incur an unnecessary performance hit from declaring another function `virtual`. This can be useful when making interfaces.

  ```
  struct Interface {
      virtual ~Interface() = 0;
  };
  Interface::~Interface() = default;

  struct Implementation : Interface {};
  // ~Implementation() is automatically defined by the compiler if not explicitly
  //  specified, meeting the "must be defined before instantiation" requirement.
  ```

- If most or all implementations of the pure virtual function will contain duplicate code, that code can instead be moved to the base class version, making the code easier to maintain.

  ```
  class SharedBase {
      State my_state;
      std::unique_ptr<Helper> my_helper;
      // ...

    public:
      virtual void config(const Context& cont) = 0;
      // ...
  };
  /* virtual */ void SharedBase::config(const Context& cont) {
      my_helper = new Helper(my_state, cont.relevant_field);
      do_this();
      and_that();
  }

  class OneImplementation : public SharedBase {
      int i;
      // ...

    public:
      void config(const Context& cont) override;
      // ...
  };
  void OneImplementation::config(const Context& cont) /* override */ {
      my_state = { cont.some_field, cont.another_field, i };
      SharedBase::config(cont);
      my_unique_setup();
  };
  ```

```
// And so on, for other classes derived from SharedBase.
```

Chapter 39: Inline functions

A function defined with the `inline` specifier is an inline function. An inline function can be multiply defined without violating the One Definition Rule, and can therefore be defined in a header with external linkage. Declaring a function inline hints to the compiler that the function should be inlined during code generation, but does not provide a guarantee.

Section 39.1: Non-member inline function definition

```
inline int add(int x, int y)
{
    return x + y;
}
```

Section 39.2: Member inline functions

```
// header (.hpp)
struct A
{
    void i_am_inlined()
    {
    }
};

struct B
{
    void i_am_NOT_inlined();
};

// source (.cpp)
void B::i_am_NOT_inlined()
{
}
```

Section 39.3: What is function inlining?

```
inline int add(int x, int y)
{
    return x + y;
}

int main()
{
    int a = 1, b = 2;
    int c = add(a, b);
}
```

In the above code, when `add` is inlined, the resulting code would become something like this

```
int main()
{
    int a = 1, b = 2;
    int c = a + b;
}
```

The inline function is nowhere to be seen, its body gets *inlined* into the caller's body. Had `add` not been inlined, a

function would be called. The overhead of calling a function -- such as creating a new stack frame, copying arguments, making local variables, jump (losing locality of reference and there by cache misses), etc. -- has to be incurred.

Section 39.4: Non-member inline function declaration

```
inline int add(int x, int y);
```

Chapter 40: Special Member Functions

Section 40.1: Default Constructor

A *default constructor* is a type of constructor that requires no parameters when called. It is named after the type it constructs and is a member function of it (as all constructors are).

```
class C{
    int i;
public:
    // the default constructor definition
    C()
    : i(0){ // member initializer list -- initialize i to 0
        // constructor function body -- can do more complex things here
    }
};

C c1; // calls default constructor of C to create object c1
C c2 = C(); // calls default constructor explicitly
C c3(); // ERROR: this intuitive version is not possible due to "most vexing parse"
C c4{}; // but in C++11 {} CAN be used in a similar way

C c5[2]; // calls default constructor for both array elements
C* c6 = new C[2]; // calls default constructor for both array elements
```

Another way to satisfy the "no parameters" requirement is for the developer to provide default values for all parameters:

```
class D{
    int i;
    int j;
public:
    // also a default constructor (can be called with no parameters)
    D( int i = 0, int j = 42 )
    : i(i), j(j){
    }
};


D d; // calls constructor of D with the provided default values for the parameters
```

Under some circumstances (i.e., the developer provides no constructors and there are no other disqualifying conditions), the compiler implicitly provides an empty default constructor:

```
class C{
    std::string s; // note: members need to be default constructible themselves
};

C c1; // will succeed -- C has an implicitly defined default constructor
```

Having some other type of constructor is one of the disqualifying conditions mentioned earlier:

```
class C{
    int i;
public:
    C( int i ) : i(i){}
};
```

```
C c1; // Compile ERROR: C has no (implicitly defined) default constructor
```

Version < c++11

To prevent implicit default constructor creation, a common technique is to declare it as `private` (with no definition). The intention is to cause a compile error when someone tries to use the constructor (this either results in an *Access to private* error or a linker error, depending on the compiler).

To be sure a default constructor (functionally similar to the implicit one) is defined, a developer could write an empty one explicitly.

Version ≥ c++11

In C++11, a developer can also use the `delete` keyword to prevent the compiler from providing a default constructor.

```
class C{
    int i;
public:
    // default constructor is explicitly deleted
    C() = delete;
};

C c1; // Compile ERROR: C has its default constructor deleted
```

Furthermore, a developer may also be explicit about wanting the compiler to provide a default constructor.

```
class C{
    int i;
public:
    // does have automatically generated default constructor (same as implicit one)
    C() = default;

    C( int i ) : i(i){}
};

C c1; // default constructed
C c2( 1 ); // constructed with the int taking constructor
```

Version ≥ c++14

You can determine whether a type has a default constructor (or is a primitive type) using `std::is_default_constructible` from **`<type_traits>`**:

```
class C1{ };
class C2{ public: C2(){} };
class C3{ public: C3(int){} };

using std::cout; using std::boolalpha; using std::endl;
using std::is_default_constructible;
cout << boolalpha << is_default_constructible<int>() << endl; // prints true
cout << boolalpha << is_default_constructible<C1>() << endl; // prints true
cout << boolalpha << is_default_constructible<C2>() << endl; // prints true
cout << boolalpha << is_default_constructible<C3>() << endl; // prints false
```

Version = c++11

In C++11, it is still possible to use the non-functor version of `std::is_default_constructible`:

```
cout << boolalpha << is_default_constructible<C1>::value << endl; // prints true
```

Section 40.2: Destructor

A *destructor* is a function without arguments that is called when a user-defined object is about to be destroyed. It is named after the type it destructs with a ~ prefix.

```
class C{
    int* is;
    string s;
public:
    C()
    : is( new int[10] ){
    }

    ~C(){  // destructor definition
        delete[] is;
    }
};

class C_child : public C{
    string s_ch;
public:
    C_child(){}
    ~C_child(){} // child destructor
};

void f(){
    C c1; // calls default constructor
    C c2[2]; // calls default constructor for both elements
    C* c3 = new C[2]; // calls default constructor for both array elements

    C_child c_ch;  // when destructed calls destructor of s_ch and of C base (and in turn s)

    delete[] c3; // calls destructors on c3[0] and c3[1]
} // automatic variables are destroyed here -- i.e. c1, c2 and c_ch
```

Under most circumstances (i.e., a user provides no destructor, and there are no other disqualifying conditions), the compiler provides a default destructor implicitly:

```
class C{
    int i;
    string s;
};

void f(){
    C* c1 = new C;
    delete c1; // C has a destructor
}
```

```
class C{
    int m;
private:
    ~C(){} // not public destructor!
};

class C_container{
    C c;
};

void f(){
```

```
    C_container* c_cont = new C_container;
    delete c_cont; // Compile ERROR: C has no accessible destructor
}
```

Version > c++11

In C++11, a developer can override this behavior by preventing the compiler from providing a default destructor.

```
class C{
    int m;
public:
    ~C() = delete; // does NOT have implicit destructor
};

void f{
    C c1;
} // Compile ERROR: C has no destructor
```

Furthermore, a developer may also be explicit about wanting the compiler to provide a default destructor.

```
class C{
    int m;
public:
    ~C() = default; // saying explicitly it does have implicit/empty destructor
};

void f(){
    C c1;
} // C has a destructor -- c1 properly destroyed
```

Version > c++11

You can determine whether a type has a destructor (or is a primitive type) using `std::is_destructible` from **<type_traits>**:

```
class C1{ };
class C2{ public: ~C2() = delete };
class C3 : public C2{ };

using std::cout; using std::boolalpha; using std::endl;
using std::is_destructible;
cout << boolalpha << is_destructible<int>() << endl; // prints true
cout << boolalpha << is_destructible<C1>() << endl; // prints true
cout << boolalpha << is_destructible<C2>() << endl; // prints false
cout << boolalpha << is_destructible<C3>() << endl; // prints false
```

Section 40.3: Copy and swap

If you're writing a class that manages resources, you need to implement all the special member functions (see Rule of Three/Five/Zero). The most direct approach to writing the copy constructor and assignment operator would be:

```
person(const person &other)
    : name(new char[std::strlen(other.name) + 1])
    , age(other.age)
{
    std::strcpy(name, other.name);
}

person& operator=(person const& rhs) {
    if (this != &other) {
        delete [] name;
```

```
        name = new char[std::strlen(other.name) + 1];
        std::strcpy(name, other.name);
        age = other.age;
    }

    return *this;
}
```

But this approach has some problems. It fails the strong exception guarantee - if `new[]` throws, we've already cleared the resources owned by `this` and cannot recover. We're duplicating a lot of the logic of copy construction in copy assignment. And we have to remember the self-assignment check, which usually just adds overhead to the copy operation, but is still critical.

To satisfy the strong exception guarantee and avoid code duplication (double so with the subsequent move assignment operator), we can use the copy-and-swap idiom:

```
class person {
    char* name;
    int age;
public:
    /* all the other functions ... */

    friend void swap(person& lhs, person& rhs) {
        using std::swap; // enable ADL

        swap(lhs.name, rhs.name);
        swap(lhs.age, rhs.age);
    }

    person& operator=(person rhs) {
        swap(*this, rhs);
        return *this;
    }
};
```

Why does this work? Consider what happens when we have

```
person p1 = ...;
person p2 = ...;
p1 = p2;
```

First, we copy-construct `rhs` from `p2` (which we didn't have to duplicate here). If that operation throws, we don't do anything in `operator=` and `p1` remains untouched. Next, we swap the members between `*this` and `rhs`, and then `rhs` goes out of scope. When `operator=`, that implicitly cleans the original resources of `this` (via the destructor, which we didn't have to duplicate). Self-assignment works too - it's less efficient with copy-and-swap (involves an extra allocation and deallocation), but if that's the unlikely scenario, we don't slow down the typical use case to account for it.

Version ≥ C++11

The above formulation works as-is already for move assignment.

```
p1 = std::move(p2);
```

Here, we move-construct `rhs` from `p2`, and all the rest is just as valid. If a class is movable but not copyable, there is no need to delete the copy-assignment, since this assignment operator will simply be ill-formed due to the deleted copy constructor.

Section 40.4: Implicit Move and Copy

Bear in mind that declaring a destructor inhibits the compiler from generating implicit move constructors and move assignment operators. If you declare a destructor, remember to also add appropriate definitions for the move operations.

Furthermore, declaring move operations will suppress the generation of copy operations, so these should also be added (if the objects of this class are required to have copy semantics).

```
class Movable {
public:
    virtual ~Movable() noexcept = default;

    //    compiler won't generate these unless we tell it to
    //    because we declared a destructor
    Movable(Movable&&) noexcept = default;
    Movable& operator=(Movable&&) noexcept = default;

    //    declaring move operations will suppress generation
    //    of copy operations unless we explicitly re-enable them
    Movable(const Movable&) = default;
    Movable& operator=(const Movable&) = default;
};
```

Chapter 41: Non-Static Member Functions

Section 41.1: Non-static Member Functions

A `class` or `struct` can have member functions as well as member variables. These functions have syntax mostly similar to standalone functions, and can be defined either inside or outside the class definition; if defined outside the class definition, the function's name is prefixed with the class' name and the scope (`::`) operator.

```
class CL {
  public:
    void  definedInside() {}
    void definedOutside();
};
void CL::definedOutside() {}
```

These functions are called on an instance (or reference to an instance) of the class with the dot (`.`) operator, or a pointer to an instance with the arrow (`->`) operator, and each call is tied to the instance the function was called on; when a member function is called on an instance, it has access to all of that instance's fields (through the `this` pointer), but can only access other instances' fields if those instances are supplied as parameters.

```
struct ST {
    ST(const std::string& ss = "Wolf", int ii = 359) : s(ss), i(ii) { }

    int get_i() const { return i; }
    bool compare_i(const ST& other) const { return (i == other.i); }

  private:
    std::string s;
    int i;
};
ST st1;
ST st2("Species", 8472);

int  i = st1.get_i(); // Can access st1.i, but not st2.i.
bool b = st1.compare_i(st2); // Can access st1 & st2.
```

These functions are allowed to access member variables and/or other member functions, regardless of either the variable or function's access modifiers. They can also be written out-of-order, accessing member variables and/or calling member functions declared before them, as the entire class definition must be parsed before the compiler can begin to compile a class.

```
class Access {
  public:
    Access(int i_ = 8088, int j_ = 8086, int k_ = 6502) : i(i_), j(j_), k(k_) {}

    int i;
    int get_k() const { return k; }
    bool private_no_more() const { return i_be_private(); }
  protected:
    int j;
    int get_i() const { return i; }
  private:
    int k;
    int get_j() const { return j; }
    bool i_be_private() const { return ((i > j) && (k < j)); }
};
```

Section 41.2: Encapsulation

A common use of member functions is for encapsulation, using an *accessor* (commonly known as a getter) and a *mutator* (commonly known as a setter) instead of accessing fields directly.

```
class Encapsulator {
    int encapsulated;

  public:
    int  get_encapsulated() const { return encapsulated; }
    void set_encapsulated(int e)  { encapsulated = e; }

    void some_func() {
        do_something_with(encapsulated);
    }
};
```

Inside the class, `encapsulated` can be freely accessed by any non-static member function; outside the class, access to it is regulated by member functions, using `get_encapsulated()` to read it and `set_encapsulated()` to modify it. This prevents unintentional modifications to the variable, as separate functions are used to read and write it. [There are many discussions on whether getters and setters provide or break encapsulation, with good arguments for both claims; such heated debate is outside the scope of this example.]

Section 41.3: Name Hiding & Importing

When a base class provides a set of overloaded functions, and a derived class adds another overload to the set, this hides all of the overloads provided by the base class.

```
struct HiddenBase {
    void f(int) { std::cout << "int" << std::endl; }
    void f(bool) { std::cout << "bool" << std::endl; }
    void f(std::string) { std::cout << "std::string" << std::endl; }
};

struct HidingDerived : HiddenBase {
    void f(float) { std::cout << "float" << std::endl; }
};

// ...

HiddenBase hb;
HidingDerived hd;
std::string s;

hb.f(1);    // Output:  int
hb.f(true); // Output:  bool
hb.f(s);    // Output:  std::string;

hd.f(1.f);  // Output:  float
hd.f(3);    // Output:  float
hd.f(true); // Output:  float
hd.f(s);    // Error: Can't convert from std::string to float.
```

This is due to name resolution rules: During name lookup, once the correct name is found, we stop looking, even if we clearly haven't found the correct *version* of the entity with that name (such as with `hd.f(s)`); due to this, overloading the function in the derived class prevents name lookup from discovering the overloads in the base class. To avoid this, a using-declaration can be used to "import" names from the base class into the derived class, so

that they will be available during name lookup.

```
struct HidingDerived : HiddenBase {
     // All members named HiddenBase::f shall be considered members of HidingDerived for lookup.
    using HiddenBase::f;

    void f(float) { std::cout << "float" << std::endl; }
};

// ...

HidingDerived hd;

hd.f(1.f);  // Output:  float
hd.f(3);    // Output:  int
hd.f(true); // Output:  bool
hd.f(s);    // Output:  std::string
```

If a derived class imports names with a using-declaration, but also declares functions with the same signature as functions in the base class, the base class functions will silently be overridden or hidden.

```
struct NamesHidden {
    virtual void hide_me()      {}
    virtual void hide_me(float) {}
    void hide_me(int)           {}
    void hide_me(bool)          {}
};

struct NameHider : NamesHidden {
    using NamesHidden::hide_me;

    void hide_me()    {} // Overrides NamesHidden::hide_me().
    void hide_me(int) {} // Hides NamesHidden::hide_me(int).
};
```

A using-declaration can also be used to change access modifiers, provided the imported entity was `public` or `protected` in the base class.

```
struct ProMem {
  protected:
    void func() {}
};

struct BecomesPub : ProMem {
    using ProMem::func;
};

// ...

ProMem pm;
BecomesPub bp;

pm.func(); // Error: protected.
bp.func(); // Good.
```

Similarly, if we explicitly want to call a member function from a specific class in the inheritance hierarchy, we can qualify the function name when calling the function, specifying that class by name.

```
struct One {
```

```
    virtual void f() { std::cout << "One." << std::endl; }
};

struct Two : One {
    void f() override {
        One::f(); // this->One::f();
        std::cout << "Two." << std::endl;
    }
};

struct Three : Two {
    void f() override {
        Two::f(); // this->Two::f();
        std::cout << "Three." << std::endl;
    }
};

// ...

Three t;

t.f();      // Normal syntax.
t.Two::f(); // Calls version of f() defined in Two.
t.One::f(); // Calls version of f() defined in One.
```

Section 41.4: Virtual Member Functions

Member functions can also be declared `virtual`. In this case, if called on a pointer or reference to an instance, they will not be accessed directly; rather, they will look up the function in the virtual function table (a list of pointers-to-member-functions for virtual functions, more commonly known as the `vtable` or `vftable`), and use that to call the version appropriate for the instance's dynamic (actual) type. If the function is called directly, from a variable of a class, no lookup is performed.

```
struct Base {
    virtual void func() { std::cout << "In Base." << std::endl; }
};

struct Derived : Base {
    void func() override { std::cout << "In Derived." << std::endl; }
};

void slicer(Base x) { x.func(); }

// ...

Base b;
Derived d;

Base *pb = &b, *pd = &d; // Pointers.
Base &rb = b, &rd = d;   // References.

b.func();   // Output:  In Base.
d.func();   // Output:  In Derived.

pb->func(); // Output:  In Base.
pd->func(); // Output:  In Derived.

rb.func();  // Output:  In Base.
rd.func();  // Output:  In Derived.
```

```
slicer(b);  // Output:  In Base.
slicer(d);  // Output:  In Base.
```

Note that while `pd` is `Base*`, and `rd` is a `Base&`, calling `func()` on either of the two calls `Derived::func()` instead of `Base::func()`; this is because the `vtable` for `Derived` updates the `Base::func()` entry to instead point to `Derived::func()`. Conversely, note how passing an instance to `slicer()` always results in `Base::func()` being called, even when the passed instance is a `Derived`; this is because of something known as *data slicing*, where passing a `Derived` instance into a `Base` parameter by value renders the portion of the `Derived` instance that isn't a `Base` instance inaccessible.

When a member function is defined as virtual, all derived class member functions with the same signature override it, regardless of whether the overriding function is specified as `virtual` or not. This can make derived classes harder for programmers to parse, however, as there's no indication as to which function(s) is/are `virtual`.

```
struct B {
    virtual void f() {}
};

struct D : B {
    void f() {} // Implicitly virtual, overrides B::f.
                //  You'd have to check B to know that, though.
};
```

Note, however, that a derived function only overrides a base function if their signatures match; even if a derived function is explicitly declared `virtual`, it will instead create a new virtual function if the signatures are mismatched.

```
struct BadB {
    virtual void f() {}
};

struct BadD : BadB {
    virtual void f(int i) {} // Does NOT override BadB::f.
};
```

Version ≥ C++11

As of C++11, intent to override can be made explicit with the context-sensitive keyword `override`. This tells the compiler that the programmer expects it to override a base class function, which causes the compiler to omit an error if it *doesn't* override anything.

```
struct CPP11B {
    virtual void f() {}
};

struct CPP11D : CPP11B {
    void f() override {}
    void f(int i) override {} // Error: Doesn't actually override anything.
};
```

This also has the benefit of telling programmers that the function is both virtual, and also declared in at least one base class, which can make complex classes easier to parse.

When a function is declared `virtual`, and defined outside the class definition, the `virtual` specifier must be included in the function declaration, and not repeated in the definition.

Version ≥ C++11

This also holds true for `override`.

```
struct VB {
    virtual void f(); // "virtual" goes here.
    void g();
};
/* virtual */ void VB::f() {} // Not here.
virtual void VB::g() {} // Error.
```

If a base class overloads a `virtual` function, only overloads that are explicitly specified as `virtual` will be virtual.

```
struct BOverload {
    virtual void func() {}
    void func(int) {}
};

struct DOverload : BOverload {
    void func() override {}
    void func(int) {}
};

// ...

BOverload* bo = new DOverload;
bo->func(); // Calls DOverload::func().
bo->func(1); // Calls BOverload::func(int).
```

For more information, see the relevant topic.

Section 41.5: Const Correctness

One of the primary uses for `this` cv-qualifiers is *`const` correctness*. This is the practice of guaranteeing that only accesses that *need* to modify an object are *able* to modify the object, and that any (member or non-member) function that doesn't need to modify an object doesn't have write access to that object (whether directly or indirectly). This prevents unintentional modifications, making code less errorprone. It also allows any function that doesn't need to modify state to be able to take either a `const` or non-`const` object, without needing to rewrite or overload the function.

`const` correctness, due to its nature, starts at the bottom up: Any class member function that doesn't need to change state is declared as `const`, so that it can be called on `const` instances. This, in turn, allows passed-by-reference parameters to be declared `const` when they don't need to be modified, which allows functions to take either `const` or non-`const` objects without complaining, and `const`-ness can propagate outwards in this manner. Due to this, getters are frequently `const`, as are any other functions that don't need to modify logical state.

```
class ConstIncorrect {
    Field fld;

  public:
    ConstIncorrect(const Field& f) : fld(f) {}     // Modifies.

    const Field& get_field()       { return fld; } // Doesn't modify; should be const.
    void set_field(const Field& f) { fld = f; }    // Modifies.

    void do_something(int i) {                     // Modifies.
        fld.insert_value(i);
    }
    void do_nothing()        { }                   // Doesn't modify; should be const.
};
```

```cpp
class ConstCorrect {
    Field fld;

  public:
    ConstCorrect(const Field& f) : fld(f) {}       // Not const: Modifies.

    const Field& get_field() const { return fld; } // const: Doesn't modify.
    void set_field(const Field& f) { fld = f; }    // Not const: Modifies.

    void do_something(int i) {                     // Not const: Modifies.
        fld.insert_value(i);
    }
    void do_nothing() const  { }                   // const: Doesn't modify.
};

// ...

const ConstIncorrect i_cant_do_anything(make_me_a_field());
// Now, let's read it...
Field f = i_cant_do_anything.get_field();
  // Error: Loses cv-qualifiers, get_field() isn't const.
i_cant_do_anything.do_nothing();
  // Error: Same as above.
// Oops.

const ConstCorrect but_i_can(make_me_a_field());
// Now, let's read it...
Field f = but_i_can.get_field(); // Good.
but_i_can.do_nothing();          // Good.
```

As illustrated by the comments on `ConstIncorrect` and `ConstCorrect`, properly cv-qualifying functions also serves as documentation. If a class is `const` correct, any function that isn't `const` can safely be assumed to change state, and any function that is `const` can safely be assumed not to change state.

Chapter 42: Constant class member functions

Section 42.1: constant member function

```
#include <iostream>
#include <map>
#include <string>

using namespace std;

class A {
public:
    map<string, string> * mapOfStrings;
public:
    A() {
        mapOfStrings = new map<string, string>();
    }

    void insertEntry(string const & key, string const & value) const {
        (*mapOfStrings)[key] = value;             // This works? Yes it does.
        delete mapOfStrings;                      // This also works
        mapOfStrings = new map<string, string>(); // This * does * not work
    }

    void refresh() {
        delete mapOfStrings;
        mapOfStrings = new map<string, string>(); // Works as refresh is non const function
    }

    void getEntry(string const & key) const {
        cout << mapOfStrings->at(key);
    }
};

int main(int argc, char* argv[]) {

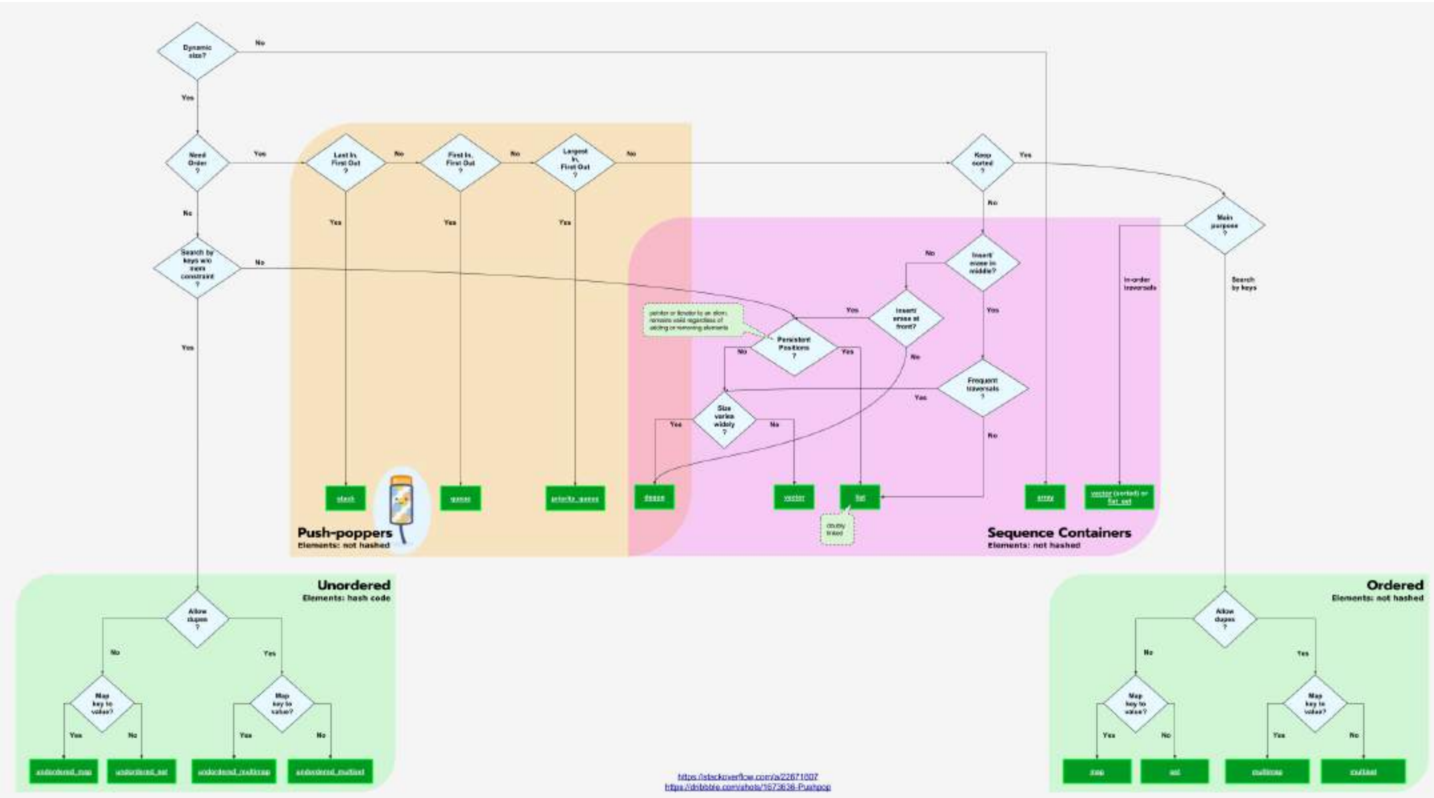
    A var;
    var.insertEntry("abc", "abcValue");
    var.getEntry("abc");
    getchar();
    return 0;
}
```

Chapter 43: C++ Containers

C++ containers store a collection of elements. Containers include vectors, lists, maps, etc. Using Templates, C++ containers contain collections of primitives (e.g. ints) or custom classes (e.g. MyClass).

Section 43.1: C++ Containers Flowchart

Choosing which C++ Container to use can be tricky, so here's a simple flowchart to help decide which Container is right for the job.



This flowchart was based on Mikael Persson's post. This little graphic in the flowchart is from Megan Hopkins

Chapter 44: Namespaces

Used to prevent name collisions when using multiple libraries, a namespace is a declarative prefix for functions, classes, types, etc.

Section 44.1: What are namespaces?

A C++ namespace is a collection of C++ entities (functions, classes, variables), whose names are prefixed by the name of the namespace. When writing code within a namespace, named entities belonging to that namespace need not be prefixed with the namespace name, but entities outside of it must use the fully qualified name. The fully qualified name has the format **`<namespace>::<entity>`**. Example:

```
namespace Example
{
  const int test = 5;

  const int test2 = test + 12; //Works within `Example` namespace
}

const int test3 = test + 3; //Fails; `test` not found outside of namespace.

const int test3 = Example::test + 3; //Works; fully qualified name used.
```

Namespaces are useful for grouping related definitions together. Take the analogy of a shopping mall. Generally a shopping mall is split up into several stores, each store selling items from a specific category. One store might sell electronics, while another store might sell shoes. These logical separations in store types help the shoppers find the items they're looking for. Namespaces help c++ programmers, like shoppers, find the functions, classes, and variables they're looking for by organizing them in a logical manner. Example:

```
namespace Electronics
{
    int TotalStock;
    class Headphones
    {
        // Description of a Headphone (color, brand, model number, etc.)
    };
    class Television
    {
        // Description of a Television (color, brand, model number, etc.)
    };
}

namespace Shoes
{
    int TotalStock;
    class Sandal
    {
        // Description of a Sandal (color, brand, model number, etc.)
    };
    class Slipper
    {
        // Description of a Slipper (color, brand, model number, etc.)
    };
}
```

There is a single namespace predefined, which is the global namespace that has no name, but can be denoted by `::`. Example:

```
void bar() {
    // defined in global namespace
}
namespace foo {
    void bar() {
        // defined in namespace foo
    }
    void barbar() {
        bar();   // calls foo::bar()
        ::bar(); // calls bar() defined in global namespace
    }
}
```

Section 44.2: Argument Dependent Lookup

When calling a function without an explicit namespace qualifier, the compiler can choose to call a function within a namespace if one of the parameter types to that function is also in that namespace. This is called "Argument Dependent Lookup", or ADL:

```
namespace Test
{
  int call(int i);

  class SomeClass {...};

  int call_too(const SomeClass &data);
}

call(5); //Fails. Not a qualified function name.

Test::SomeClass data;

call_too(data); //Succeeds
```

`call` fails because none of its parameter types come from the `Test` namespace. `call_too` works because `SomeClass` is a member of `Test` and therefore it qualifies for ADL rules.

When does ADL not occur

ADL does not occur if normal unqualified lookup finds a class member, a function that has been declared at block scope, or something that is not of function type. For example:

```
void foo();
namespace N {
    struct X {};
    void foo(X ) { std::cout << '1'; }
    void qux(X ) { std::cout << '2'; }
}

struct C {
    void foo() {}
    void bar() {
        foo(N::X{}); // error: ADL is disabled and C::foo() takes no arguments
    }
};

void bar() {
    extern void foo(); // redeclares ::foo
    foo(N::X{});       // error: ADL is disabled and ::foo() doesn't take any arguments
```

```
}

int qux;

void baz() {
    qux(N::X{}); // error: variable declaration disables ADL for "qux"
}
```

Section 44.3: Extending namespaces

A useful feature of `namespace`s is that you can expand them (add members to it).

```
namespace Foo
{
    void bar() {}
}

//some other stuff

namespace Foo
{
    void bar2() {}
}
```

Section 44.4: Using directive

The keyword 'using' has three flavors. Combined with keyword 'namespace' you write a 'using directive':

If you don't want to write `Foo::` in front of every stuff in the namespace `Foo`, you can use `using namespace Foo;` to import every single thing out of `Foo`.

```
namespace Foo
{
    void bar() {}
    void baz() {}
}

//Have to use Foo::bar()
Foo::bar();

//Import Foo
using namespace Foo;
bar(); //OK
baz(); //OK
```

It is also possible to import selected entities in a namespace rather than the whole namespace:

```
using Foo::bar;
bar(); //OK, was specifically imported
baz(); // Not OK, was not imported
```

A word of caution: `using namespace`s in header files is seen as bad style in most cases. If this is done, the namespace is imported in *every* file that includes the header. Since there is no way of "un-`using`" a namespace, this can lead to namespace pollution (more or unexpected symbols in the global namespace) or, worse, conflicts. See this example for an illustration of the problem:

```
/***** foo.h *****/
```

```
namespace Foo
{
    class C;
}

/***** bar.h *****/
namespace Bar
{
    class C;
}

/***** baz.h *****/
#include "foo.h"
using namespace Foo;

/***** main.cpp *****/
#include "bar.h"
#include "baz.h"

using namespace Bar;
C c; // error: Ambiguity between Bar::C and Foo::C
```

A *using-directive* cannot occur at class scope.

Section 44.5: Making namespaces

Creating a namespace is really easy:

```
//Creates namespace foo
namespace Foo
{
    //Declares function bar in namespace foo
    void bar() {}
}
```

To call `bar`, you have to specify the namespace first, followed by the scope resolution operator `::`:

```
Foo::bar();
```

It is allowed to create one namespace in another, for example:

```
namespace A
{
    namespace B
    {
        namespace C
        {
            void bar() {}
        }
    }
}
```

Version ≥ C++17

The above code could be simplified to the following:

```
namespace A::B::C
{
    void bar() {}
```

```
}
```

Section 44.6: Unnamed/anonymous namespaces

An unnamed namespace can be used to ensure names have internal linkage (can only be referred to by the current translation unit). Such a namespace is defined in the same way as any other namespace, but without the name:

```
namespace {
    int foo = 42;
}
```

`foo` is only visible in the translation unit in which it appears.

It is recommended to never use unnamed namespaces in header files as this gives a version of the content for every translation unit it is included in. This is especially important if you define non-const globals.

```
// foo.h
namespace {
    std::string globalString;
}

// 1.cpp
#include "foo.h" //< Generates unnamed_namespace{1.cpp}::globalString ...

globalString = "Initialize";

// 2.cpp
#include "foo.h" //< Generates unnamed_namespace{2.cpp}::globalString ...

std::cout << globalString; //< Will always print the empty string
```

Section 44.7: Compact nested namespaces

Version ≥ C++17

```
namespace a {
  namespace b {
    template<class T>
    struct qualifies : std::false_type {};
  }
}

namespace other {
  struct bob {};
}

namespace a::b {
  template<>
  struct qualifies<::other::bob> : std::true_type {};
}
```

You can enter both the `a` and `b` namespaces in one step with `namespace a::b` starting in C++17.

Section 44.8: Namespace alias

A namespace can be given an alias (*i.e.,* another name for the same namespace) using the `namespace` *identifier* = syntax. Members of the aliased namespace can be accessed by qualifying them with the name of the alias. In the

following example, the nested namespace `AReallyLongName::AnotherReallyLongName` is inconvenient to type, so the function `qux` locally declares an alias `N`. Members of that namespace can then be accessed simply using `N::`.

```
namespace AReallyLongName {
    namespace AnotherReallyLongName {
        int foo();
        int bar();
        void baz(int x, int y);
    }
}
void qux() {
    namespace N = AReallyLongName::AnotherReallyLongName;
    N::baz(N::foo(), N::bar());
}
```

Section 44.9: Inline namespace

Version ≥ C++11

`inline namespace` includes the content of the inlined namespace in the enclosing namespace, so

```
namespace Outer
{
    inline namespace Inner
    {
        void foo();
    }
}
```

is mostly equivalent to

```
namespace Outer
{

    namespace Inner
    {
        void foo();
    }

    using Inner::foo;
}
```

but element from `Outer::Inner::` and those associated into `Outer::` are identical.

So following is equivalent

```
Outer::foo();
Outer::Inner::foo();
```

The alternative `using namespace Inner;` would not be equivalent for some tricky parts as template specialization:

For

```
#include <outer.h> // See below

class MyCustomType;
namespace Outer
{
```

```
    template <>
    void foo<MyCustomType>() { std::cout << "Specialization"; }
}
```

- The inline namespace allows the specialization of `Outer::foo`

```
// outer.h
// include guard omitted for simplification

namespace Outer
{
    inline namespace Inner
    {
        template <typename T>
        void foo() { std::cout << "Generic"; }
    }
}
```

- Whereas the `using namespace` doesn't allow the specialization of `Outer::foo`

```
// outer.h
// include guard omitted for simplification

namespace Outer
{
    namespace Inner
    {
        template <typename T>
        void foo() { std::cout << "Generic"; }
    }
    using namespace Inner;
    // Specialization of `Outer::foo` is not possible
    // it should be `Outer::Inner::foo`.
}
```

Inline namespace is a way to allow several version to cohabit and defaulting to the `inline` one

```
namespace MyNamespace
{
    // Inline the last version
    inline namespace Version2
    {
        void foo(); // New version
        void bar();
    }

    namespace Version1 // The old one
    {
        void foo();
    }

}
```

And with usage

```
MyNamespace::Version1::foo(); // old version
MyNamespace::Version2::foo(); // new version
MyNamespace::foo();           // default version : MyNamespace::Version1::foo();
```

Section 44.10: Aliasing a long namespace

This is usually used for renaming or shortening long namespace references such referring to components of a library.

```
namespace boost
{
    namespace multiprecision
    {
        class Number ...
    }
}

namespace Name1 = boost::multiprecision;


//    Both Type declarations are equivalent
boost::multiprecision::Number X   //    Writing the full namespace path, longer
Name1::Number Y                   //    using the name alias, shorter
```

Section 44.11: Alias Declaration scope

Alias Declaration are affected by preceding *using* statements

```
namespace boost
{
    namespace multiprecision
    {
        class Number ...
    }
}


using namespace boost;

//   Both Namespace are equivalent
namespace Name1 = boost::multiprecision;
namespace Name2 = multiprecision;
```

However, it is easier to get confused over which namespace you are aliasing when you have something like this:

```
namespace boost
{
    namespace multiprecision
    {
        class Number ...
    }
}

namespace numeric
{
    namespace multiprecision
    {
        class Number ...
    }
}

using namespace numeric;
using namespace boost;
```

```
//    Not recommended as
//    its not explicitly clear whether Name1 refers to
//    numeric::multiprecision or boost::multiprecision
namespace Name1 = multiprecision;

//    For clarity, its recommended to use absolute paths
//    instead
namespace Name2 = numeric::multiprecision;
namespace Name3 = boost::multiprecision;
```

Chapter 45: Header Files

Section 45.1: Basic Example

The following example will contain a block of code that is meant to be split into several source files, as denoted by `// filename` comments.

Source Files

```
// my_function.h

/* Note how this header contains only a declaration of a function.
 * Header functions usually do not define implementations for declarations
 * unless code must be further processed at compile time, as in templates.
 */

/* Also, usually header files include preprocessor guards so that every header
 * is never included twice.
 *
 * The guard is implemented by checking if a header-file unique preprocessor
 * token is defined, and only including the header if it hasn't been included
 * once before.
 */
#ifndef MY_FUNCTION_H
#define MY_FUNCTION_H

// global_value and my_function() will be
// recognized as the same constructs if this header is included by different files.
const int global_value = 42;
int my_function();

#endif // MY_FUNCTION_H
```

```
// my_function.cpp

/* Note how the corresponding source file for the header includes the interface
 * defined in the header so that the compiler is aware of what the source file is
 * implementing.
 *
 * In this case, the source file requires knowledge of the global constant
 * global_value only defined in my_function.h. Without inclusion of the header
 * file, this source file would not compile.
 */
#include "my_function.h" // or #include "my_function.hpp"
int my_function() {
  return global_value; // return 42;
}
```

Header files are then included by other source files that want to use the functionality defined by the header interface, but don't require knowledge of its implementation (thus, reducing code coupling). The following program makes use of the header `my_function.h` as defined above:

```
// main.cpp

#include <iostream>        // A C++ Standard Library header.
#include "my_function.h"   // A personal header

int main(int argc, char** argv) {
  std::cout << my_function() << std::endl;
```

```
  return 0;
}
```

The Compilation Process

Since header files are often part of a compilation process workflow, a typical compilation process making use of the header/source file convention will usually do the following.

Assuming that the header file and source code file is already in the same directory, a programmer would execute the following commands:

```
g++ -c my_function.cpp       # Compiles the source file my_function.cpp
                             # --> object file my_function.o

g++ main.cpp my_function.o   # Links the object file containing the
                             # implementation of int my_function()
                             # to the compiled, object version of main.cpp
                             # and then produces the final executable a.out
```

Alternatively, if one wishes to compile `main.cpp` to an object file first, and then link only object files together as the final step:

```
g++ -c my_function.cpp
g++ -c main.cpp

g++ main.o my_function.o
```

Section 45.2: Templates in Header Files

Templates require compile-time generation of code: a templated function, for example, will be effectively turned into multiple distinct functions once a templated function is parameterized by use in source code.

This means that template function, member function, and class definitions cannot be delegated to a separate source code file, as any code that will use any templated construct requires knowledge of its definition to generally generate any derivative code.

Thus, templated code, if put in headers, must also contain its definition. An example of this is below:

```
// templated_function.h

template <typename T>
T* null_T_pointer() {
  T* type_point = NULL; // or, alternatively, nullptr instead of NULL
                        // for C++11 or later
  return type_point;
}
```

Chapter 46: Using declaration

A `using` declaration introduces a single name into the current scope that was previously declared elsewhere.

Section 46.1: Importing names individually from a namespace

Once `using` is used to introduce the name `cout` from the namespace `std` into the scope of the `main` function, the `std::cout` object can be referred to as `cout` alone.

```
#include <iostream>
int main() {
    using std::cout;
    cout << "Hello, world!\n";
}
```

Section 46.2: Redeclaring members from a base class to avoid name hiding

If a *using-declaration* occurs at class scope, it is only allowed to redeclare a member of a base class. For example, `using std::cout` is not allowed at class scope.

Often, the name redeclared is one that would otherwise be hidden. For example, in the below code, `d1.foo` only refers to `Derived1::foo(const char*)` and a compilation error will occur. The function `Base::foo(int)` is hidden not considered at all. However, `d2.foo(42)` is fine because the *using-declaration* brings `Base::foo(int)` into the set of entities named `foo` in `Derived2`. Name lookup then finds both `foo`s and overload resolution selects `Base::foo`.

```
struct Base {
    void foo(int);
};
struct Derived1 : Base {
    void foo(const char*);
};
struct Derived2 : Base {
    using Base::foo;
    void foo(const char*);
};
int main() {
    Derived1 d1;
    d1.foo(42);  // error
    Derived2 d2;
    d2.foo(42);  // OK
}
```

Section 46.3: Inheriting constructors

Version ≥ C++11

As a special case, a *using-declaration* at class scope can refer to the constructors of a direct base class. Those constructors are then *inherited* by the derived class and can be used to initialize the derived class.

```
struct Base {
    Base(int x, const char* s);
};
struct Derived1 : Base {
    Derived1(int x, const char* s) : Base(x, s) {}
};
```

```
struct Derived2 : Base {
    using Base::Base;
};
int main() {
    Derived1 d1(42, "Hello, world");
    Derived2 d2(42, "Hello, world");
}
```

In the above code, both `Derived1` and `Derived2` have constructors that forward the arguments directly to the corresponding constructor of `Base`. `Derived1` performs the forwarding explicitly, while `Derived2`, using the C++11 feature of inheriting constructors, does so implicitly.

Chapter 47: std::string

Strings are objects that represent sequences of characters. The standard `string` class provides a simple, safe and versatile alternative to using explicit arrays of `char`s when dealing with text and other sequences of characters. The C++ `string` class is part of the `std` namespace and was standardized in 1998.

Section 47.1: Tokenize

Listed from least expensive to most expensive at run-time:

1. `std::strtok` is the cheapest standard provided tokenization method, it also allows the delimiter to be modified between tokens, but it incurs 3 difficulties with modern C++:

    - `std::strtok` cannot be used on multiple `strings` at the same time (though some implementations do extend to support this, such as: `strtok_s`)
    - For the same reason `std::strtok` cannot be used on multiple threads simultaneously (this may however be implementation defined, for example: Visual Studio's implementation is thread safe)
    - Calling `std::strtok` modifies the `std::string` it is operating on, so it cannot be used on `const strings`, `const char*`s, or literal strings, to tokenize any of these with `std::strtok` or to operate on a `std::string` who's contents need to be preserved, the input would have to be copied, then the copy could be operated on

    Generally any of these options cost will be hidden in the allocation cost of the tokens, but if the cheapest algorithm is required and `std::strtok`'s difficulties are not overcomable consider a hand-spun solution.

```
// String to tokenize
std::string str{ "The quick brown fox" };
// Vector to store tokens
vector<std::string> tokens;

for (auto i = strtok(&str[0], " "); i != NULL; i = strtok(NULL, " "))
    tokens.push_back(i);
```

Live Example

2. The `std::istream_iterator` uses the stream's extraction operator iteratively. If the input `std::string` is white-space delimited this is able to expand on the `std::strtok` option by eliminating its difficulties, allowing inline tokenization thereby supporting the generation of a `const vector<string>`, and by adding support for multiple delimiting white-space character:

```
// String to tokenize
const std::string str("The  quick \tbrown \nfox");
std::istringstream is(str);
// Vector to store tokens
const std::vector<std::string> tokens = std::vector<std::string>(
                                        std::istream_iterator<std::string>(is),
                                        std::istream_iterator<std::string>());
```

Live Example

3. The `std::regex_token_iterator` uses a `std::regex` to iteratively tokenize. It provides for a more flexible delimiter definition. For example, non-delimited commas and white-space:

Version ≥ C++11

```
// String to tokenize
const std::string str{ "The ,qu\\,ick ,\tbrown, fox" };
const std::regex re{ "\\s*((?:[^\\\\,]|\\\\.)*?)\\s*(?:,|$)" };
// Vector to store tokens
const std::vector<std::string> tokens{
    std::sregex_token_iterator(str.begin(), str.end(), re, 1),
    std::sregex_token_iterator()
};
```

Live Example

See the `regex_token_iterator` Example for more details.

Section 47.2: Conversion to (const) char*

In order to get `const char*` access to the data of a `std::string` you can use the string's `c_str()` member function. Keep in mind that the pointer is only valid as long as the `std::string` object is within scope and remains unchanged, that means that only `const` methods may be called on the object.

Version ≥ C++17

The `data()` member function can be used to obtain a modifiable `char*`, which can be used to manipulate the `std::string` object's data.

Version ≥ C++11

A modifiable `char*` can also be obtained by taking the address of the first character: `&s[0]`. Within C++11, this is guaranteed to yield a well-formed, null-terminated string. Note that `&s[0]` is well-formed even if `s` is empty, whereas `&s.front()` is undefined if `s` is empty.

Version ≥ C++11

```
std::string str("This is a string.");
const char* cstr = str.c_str(); // cstr points to: "This is a string.\0"
const char* data = str.data();  // data points to: "This is a string.\0"
```

```
std::string str("This is a string.");

// Copy the contents of str to untie lifetime from the std::string object
std::unique_ptr<char []> cstr = std::make_unique<char[]>(str.size() + 1);

// Alternative to the line above (no exception safety):
// char* cstr_unsafe = new char[str.size() + 1];

std::copy(str.data(), str.data() + str.size(), cstr);
cstr[str.size()] = '\0'; // A null-terminator needs to be added

// delete[] cstr_unsafe;
std::cout << cstr.get();
```

Section 47.3: Using the std::string_view class

Version ≥ C++17

C++17 introduces `std::string_view`, which is simply a non-owning range of `const char`s, implementable as either a pair of pointers or a pointer and a length. It is a superior parameter type for functions that requires non-modifiable string data. Before C++17, there were three options for this:

```
void foo(std::string const& s);      // pre-C++17, single argument, could incur
```

```
                                  // allocation if caller's data was not in a string
                                  // (e.g. string literal or vector<char> )

void foo(const char* s, size_t len); // pre-C++17, two arguments, have to pass them
                                  // both everywhere

void foo(const char* s);          // pre-C++17, single argument, but need to call
                                  // strlen()

template <class StringT>
void foo(StringT const& s);       // pre-C++17, caller can pass arbitrary char data
                                  // provider, but now foo() has to live in a header
```

All of these can be replaced with:

```
void foo(std::string_view s);     // post-C++17, single argument, tighter coupling
                                  // zero copies regardless of how caller is storing
                                  // the data
```

*Note that `std::string_view` **cannot** modify its underlying data.*

`string_view` is useful when you want to avoid unnecessary copies.

It offers a useful subset of the functionality that `std::string` does, although some of the functions behave differently:

```
std::string str = "lllloooonnnngggg sssstttrrriiinnnggg"; //A really long string

//Bad way - 'string::substr' returns a new string (expensive if the string is long)
std::cout << str.substr(15, 10) << '\n';

//Good way - No copies are created!
std::string_view view = str;

// string_view::substr returns a new string_view
std::cout << view.substr(15, 10) << '\n';
```

Section 47.4: Conversion to std::wstring

In C++, sequences of characters are represented by specializing the `std::basic_string` class with a native character type. The two major collections defined by the standard library are `std::string` and `std::wstring`:

- `std::string` is built with elements of type `char`
- `std::wstring` is built with elements of type `wchar_t`

To convert between the two types, use `wstring_convert`:

```
#include <string>
#include <codecvt>
#include <locale>

std::string input_str = "this is a -string-, which is a sequence based on the -char- type.";
std::wstring input_wstr = L"this is a -wide- string, which is based on the -wchar_t- type.";

// conversion
std::wstring str_turned_to_wstr =
std::wstring_convert<std::codecvt_utf8<wchar_t>>().from_bytes(input_str);
```

```
std::string wstr_turned_to_str =
std::wstring_convert<std::codecvt_utf8<wchar_t>>().to_bytes(input_wstr);
```

In order to improve usability and/or readability, you can define functions to perform the conversion:

```
#include <string>
#include <codecvt>
#include <locale>

using convert_t = std::codecvt_utf8<wchar_t>;
std::wstring_convert<convert_t, wchar_t> strconverter;

std::string to_string(std::wstring wstr)
{
    return strconverter.to_bytes(wstr);
}

std::wstring to_wstring(std::string str)
{
    return strconverter.from_bytes(str);
}
```

Sample usage:

```
std::wstring a_wide_string = to_wstring("Hello World!");
```

That's certainly more readable than `std::wstring_convert<std::codecvt_utf8<wchar_t>>().from_bytes("Hello World!")`.

Please note that `char` and `wchar_t` do not imply encoding, and gives no indication of size in bytes. For instance, `wchar_t` is commonly implemented as a 2-bytes data type and typically contains UTF-16 encoded data under Windows (or UCS-2 in versions prior to Windows 2000) and as a 4-bytes data type encoded using UTF-32 under Linux. This is in contrast with the newer types `char16_t` and `char32_t`, which were introduced in C++11 and are guaranteed to be large enough to hold any UTF16 or UTF32 "character" (or more precisely, *code point*) respectively.

Section 47.5: Lexicographical comparison

Two `std::string`s can be compared lexicographically using the operators `==`, `!=`, `<`, `<=`, `>`, and `>=`:

```
std::string str1 = "Foo";
std::string str2 = "Bar";

assert(!(str1 < str2));
assert(str > str2);
assert(!(str1 <= str2));
assert(str1 >= str2);
assert(!(str1 == str2));
assert(str1 != str2);
```

All these functions use the underlying `std::string::compare()` method to perform the comparison, and return for convenience boolean values. The operation of these functions may be interpreted as follows, regardless of the actual implementation:

- operator`==`:

  If `str1.length() == str2.length()` and each character pair matches, then returns `true`, otherwise returns

`false`.

- operator`!=`:

  If `str1.length() != str2.length()` or one character pair doesn't match, returns `true`, otherwise it returns `false`.

- operator`<` or operator`>`:

  Finds the first different character pair, compares them then returns the boolean result.

- operator`<=` or operator`>=`:

  Finds the first different character pair, compares them then returns the boolean result.

Note: The term **character pair** means the corresponding characters in both strings of the same positions. For better understanding, if two example strings are `str1` and `str2`, and their lengths are `n` and `m` respectively, then character pairs of both strings means each `str1[i]` and `str2[i]` pairs where *i = 0, 1, 2, ..., max(n,m)*. If for any *i* where the corresponding character does not exist, that is, when *i* is greater than or equal to `n` or `m`, it would be considered as the lowest value.

Here is an example of using `<`:

```
std::string str1 = "Barr";
std::string str2 = "Bar";

assert(str2 < str1);
```

The steps are as follows:

1. Compare the first characters, `'B' == 'B'` - move on.
2. Compare the second characters, `'a' == 'a'` - move on.
3. Compare the third characters, `'r' == 'r'` - move on.
4. The `str2` range is now exhausted, while the `str1` range still has characters. Thus, `str2 < str1`.

Section 47.6: Trimming characters at start/end

This example requires the headers **`<algorithm>`**, **`<locale>`**, and **`<utility>`**.

Version ≥ C++11

To *trim* a sequence or string means to remove all leading and trailing elements (or characters) matching a certain predicate. We first trim the trailing elements, because it doesn't involve moving any elements, and then trim the leading elements. Note that the generalizations below work for all types of `std::basic_string` (e.g. `std::string` and `std::wstring`), and accidentally also for sequence containers (e.g. `std::vector` and `std::list`).

```
template <typename Sequence, // any basic_string, vector, list etc.
          typename Pred>     // a predicate on the element (character) type
Sequence& trim(Sequence& seq, Pred pred) {
    return trim_start(trim_end(seq, pred), pred);
}
```

Trimming the trailing elements involves finding the *last* element not matching the predicate, and erasing from there on:

```
template <typename Sequence, typename Pred>
Sequence& trim_end(Sequence& seq, Pred pred) {
    auto last = std::find_if_not(seq.rbegin(),
                                 seq.rend(),
                                 pred);
    seq.erase(last.base(), seq.end());
    return seq;
}
```

Trimming the leading elements involves finding the *first* element not matching the predicate and erasing up to there:

```
template <typename Sequence, typename Pred>
Sequence& trim_start(Sequence& seq, Pred pred) {
    auto first = std::find_if_not(seq.begin(),
                                  seq.end(),
                                  pred);
    seq.erase(seq.begin(), first);
    return seq;
}
```

To specialize the above for trimming whitespace in a `std::string` we can use the `std::isspace()` function as a predicate:

```
std::string& trim(std::string& str, const std::locale& loc = std::locale()) {
    return trim(str, [&loc](const char c){ return std::isspace(c, loc); });
}

std::string& trim_start(std::string& str, const std::locale& loc = std::locale()) {
    return trim_start(str, [&loc](const char c){ return std::isspace(c, loc); });
}

std::string& trim_end(std::string& str, const std::locale& loc = std::locale()) {
    return trim_end(str, [&loc](const char c){ return std::isspace(c, loc); });
}
```

Similarly, we can use the `std::iswspace()` function for `std::wstring` etc.

If you wish to create a *new* sequence that is a trimmed copy, then you can use a separate function:

```
template <typename Sequence, typename Pred>
Sequence trim_copy(Sequence seq, Pred pred) { // NOTE: passing seq by value
    trim(seq, pred);
    return seq;
}
```

Section 47.7: String replacement

Replace by position

To replace a portion of a `std::string` you can use the method `replace` from `std::string`.

`replace` has a lot of useful overloads:

```
//Define string
std::string str = "Hello foo, bar and world!";
std::string alternate = "Hello foobar";

//1)
str.replace(6, 3, "bar"); //"Hello bar, bar and world!"

//2)
str.replace(str.begin() + 6, str.end(), "nobody!"); //"Hello nobody!"

//3)
str.replace(19, 5, alternate, 6, 6); //"Hello foo, bar and foobar!"
```

Version ≥ C++14

```
//4)
str.replace(19, 5, alternate, 6); //"Hello foo, bar and foobar!"
```

```
//5)
str.replace(str.begin(), str.begin() + 5, str.begin() + 6, str.begin() + 9);
//"foo foo, bar and world!"

//6)
str.replace(0, 5, 3, 'z'); //"zzz foo, bar and world!"

//7)
str.replace(str.begin() + 6, str.begin() + 9, 3, 'x'); //"Hello xxx, bar and world!"
```

Version ≥ C++11

```
//8)
str.replace(str.begin(), str.begin() + 5, { 'x', 'y', 'z' }); //"xyz foo, bar and world!"
```

Replace occurrences of a string with another string

Replace only the first occurrence of `replace` with `with` in `str`:

```
std::string replaceString(std::string str,
                          const std::string& replace,
                          const std::string& with){
    std::size_t pos = str.find(replace);
    if (pos != std::string::npos)
        str.replace(pos, replace.length(), with);
    return str;
}
```

Replace all occurrence of `replace` with `with` in `str`:

```
std::string replaceStringAll(std::string str,
                             const std::string& replace,
                             const std::string& with) {
    if(!replace.empty()) {
        std::size_t pos = 0;
        while ((pos = str.find(replace, pos)) != std::string::npos) {
            str.replace(pos, replace.length(), with);
            pos += with.length();
        }
    }
    return str;
}
```

Section 47.8: Converting to std::string

`std::ostringstream` can be used to convert any streamable type to a string representation, by inserting the object

into a `std::ostringstream` object (with the stream insertion operator `<<`) and then converting the whole `std::ostringstream` to a `std::string`.

For `int` for instance:

```
#include <sstream>

int main()
{
    int val = 4;
    std::ostringstream str;
    str << val;
    std::string converted = str.str();
    return 0;
}
```

Writing your own conversion function, the simple:

```
template<class T>
std::string toString(const T& x)
{
  std::ostringstream ss;
  ss << x;
  return ss.str();
}
```

works but isn't suitable for performance critical code.

User-defined classes may implement the stream insertion operator if desired:

```
std::ostream operator<<( std::ostream& out, const A& a )
{
    // write a string representation of a to out
    return out;
}
```

Version ≥ C++11

Aside from streams, since C++11 you can also use the `std::to_string` (and `std::to_wstring`) function which is overloaded for all fundamental types and returns the string representation of its parameter.

```
std::string s = to_string(0x12f3);  // after this the string s contains "4851"
```

Section 47.9: Splitting

Use `std::string::substr` to split a string. There are two variants of this member function.

The first takes a *starting position* from which the returned substring should begin. The starting position must be valid in the range `(0, str.length()]`:

```
std::string str = "Hello foo, bar and world!";
std::string newstr = str.substr(11); // "bar and world!"
```

The second takes a starting position and a total *length* of the new substring. Regardless of the *length*, the substring will never go past the end of the source string:

```
std::string str = "Hello foo, bar and world!";
```

```
std::string newstr = str.substr(15, 3); // "and"
```

Note that you can also call `substr` with no arguments, in this case an exact copy of the string is returned

```
std::string str = "Hello foo, bar and world!";
std::string newstr = str.substr(); // "Hello foo, bar and world!"
```

Section 47.10: Accessing a character

There are several ways to extract characters from a `std::string` and each is subtly different.

```
std::string str("Hello world!");
```

operator[](n)

Returns a reference to the character at index n.

`std::string::operator[]` is not bounds-checked and does not throw an exception. The caller is responsible for asserting that the index is within the range of the string:

```
char c = str[6]; // 'w'
```

at(n)

Returns a reference to the character at index n.

`std::string::at` *is* bounds checked, and will throw `std::out_of_range` if the index is not within the range of the string:

```
char c = str.at(7); // 'o'
```

Version ≥ C++11

Note: Both of these examples will result in undefined behavior if the string is empty.

front()

Returns a reference to the first character:

```
char c = str.front(); // 'H'
```

back()

Returns a reference to the last character:

```
char c = str.back(); // '!'
```

Section 47.11: Checking if a string is a prefix of another

Version ≥ C++14

In C++14, this is easily done by `std::mismatch` which returns the first mismatching pair from two ranges:

```
std::string prefix = "foo";
```

```
std::string string = "foobar";

bool isPrefix = std::mismatch(prefix.begin(), prefix.end(),
    string.begin(), string.end()).first == prefix.end();
```

Note that a range-and-a-half version of `mismatch()` existed prior to C++14, but this is unsafe in the case that the second string is the shorter of the two.

Version < C++14

We can still use the range-and-a-half version of `std::mismatch()`, but we need to first check that the first string is at most as big as the second:

```
bool isPrefix = prefix.size() <= string.size() &&
    std::mismatch(prefix.begin(), prefix.end(),
        string.begin(), string.end()).first == prefix.end();
```

Version ≥ C++17

With `std::string_view`, we can write the direct comparison we want without having to worry about allocation overhead or making copies:

```
bool isPrefix(std::string_view prefix, std::string_view full)
{
    return prefix == full.substr(0, prefix.size());
}
```

Section 47.12: Looping through each character

Version ≥ C++11

`std::string` supports iterators, and so you can use a *ranged based* loop to iterate through each character:

```
std::string str = "Hello World!";
for (auto c : str)
    std::cout << c;
```

You can use a "traditional" `for` loop to loop through every character:

```
std::string str = "Hello World!";
for (std::size_t i = 0; i < str.length(); ++i)
    std::cout << str[i];
```

Section 47.13: Conversion to integers/floating point types

A `std::string` containing a number can be converted into an integer type, or a floating point type, using conversion functions.

Note that all of these functions stop parsing the input string as soon as they encounter a non-numeric character, so `"123abc"` will be converted into `123`.

The `std::ato*` family of functions converts C-style strings (character arrays) to integer or floating-point types:

```
std::string ten = "10";

double num1 = std::atof(ten.c_str());
```

```
int num2 = std::atoi(ten.c_str());
long num3 = std::atol(ten.c_str());
```

Version ≥ C++11

```
long long num4 = std::atoll(ten.c_str());
```

However, use of these functions is discouraged because they return `0` if they fail to parse the string. This is bad because `0` could also be a valid result, if for example the input string was "0", so it is impossible to determine if the conversion actually failed.

The newer `std::sto*` family of functions convert `std::string`s to integer or floating-point types, and throw exceptions if they could not parse their input. *You should use these functions if possible*:

Version ≥ C++11

```
std::string ten = "10";

int num1 = std::stoi(ten);
long num2 = std::stol(ten);
long long num3 = std::stoll(ten);

float num4 = std::stof(ten);
double num5 = std::stod(ten);
long double num6 = std::stold(ten);
```

Furthermore, these functions also handle octal and hex strings unlike the `std::ato*` family. The second parameter is a pointer to the first unconverted character in the input string (not illustrated here), and the third parameter is the base to use. `0` is automatic detection of octal (starting with `0`) and hex (starting with `0x` or `0X`), and any other value is the base to use

```
std::string ten = "10";
std::string ten_octal = "12";
std::string ten_hex = "0xA";

int num1 = std::stoi(ten, 0, 2); // Returns 2
int num2 = std::stoi(ten_octal, 0, 8); // Returns 10
long num3 = std::stol(ten_hex, 0, 16);  // Returns 10
long num4 = std::stol(ten_hex);  // Returns 0
long num5 = std::stol(ten_hex, 0, 0); // Returns 10 as it detects the leading 0x
```

Section 47.14: Concatenation

You can concatenate `std::string`s using the overloaded `+` and `+=` operators. Using the `+` operator:

```
std::string hello = "Hello";
std::string world = "world";
std::string helloworld = hello + world; // "Helloworld"
```

Using the `+=` operator:

```
std::string hello = "Hello";
std::string world = "world";
hello += world; // "Helloworld"
```

You can also append C strings, including string literals:

```
std::string hello = "Hello";
std::string world = "world";
```

```
const char *comma = ", ";
std::string newhelloworld = hello + comma + world + "!"; // "Hello, world!"
```

You can also use `push_back()` to push back individual `char`s:

```
std::string s = "a, b, ";
s.push_back('c'); // "a, b, c"
```

There is also `append()`, which is pretty much like `+=`:

```
std::string app = "test and ";
app.append("test"); // "test and test"
```

Section 47.15: Converting between character encodings

Converting between encodings is easy with C++11 and most compilers are able to deal with it in a cross-platform manner through **`<codecvt>`** and **`<locale>`** headers.

```
#include <iostream>
#include <codecvt>
#include <locale>
#include <string>
using namespace std;

int main() {
    // converts between wstring and utf8 string
    wstring_convert<codecvt_utf8_utf16<wchar_t>> wchar_to_utf8;
    // converts between u16string and utf8 string
    wstring_convert<codecvt_utf8_utf16<char16_t>, char16_t> utf16_to_utf8;

    wstring wstr = L"foobar";
    string utf8str = wchar_to_utf8.to_bytes(wstr);
    wstring wstr2 = wchar_to_utf8.from_bytes(utf8str);

    wcout << wstr << endl;
    cout << utf8str << endl;
    wcout << wstr2 << endl;

    u16string u16str = u"foobar";
    string utf8str2 = utf16_to_utf8.to_bytes(u16str);
    u16string u16str2 = utf16_to_utf8.from_bytes(utf8str2);

    return 0;
}
```

Mind that Visual Studio 2015 provides supports for these conversion but a bug in their library implementation requires to use a different template for `wstring_convert` when dealing with `char16_t`:

```
using utf16_char = unsigned short;
wstring_convert<codecvt_utf8_utf16<utf16_char>, utf16_char> conv_utf8_utf16;

void strings::utf16_to_utf8(const std::u16string& utf16, std::string& utf8)
{
  std::basic_string<utf16_char> tmp;
  tmp.resize(utf16.length());
  std::copy(utf16.begin(), utf16.end(), tmp.begin());
  utf8 = conv_utf8_utf16.to_bytes(tmp);
}
```

```
void strings::utf8_to_utf16(const std::string& utf8, std::u16string& utf16)
{
  std::basic_string<utf16_char> tmp = conv_utf8_utf16.from_bytes(utf8);
  utf16.clear();
  utf16.resize(tmp.length());
  std::copy(tmp.begin(), tmp.end(), utf16.begin());
}
```

Section 47.16: Finding character(s) in a string

To find a character or another string, you can use `std::string::find`. It returns the position of the first character of the first match. If no matches were found, the function returns `std::string::npos`

```
std::string str = "Curiosity killed the cat";
auto it = str.find("cat");

if (it != std::string::npos)
    std::cout << "Found at position: " << it << '\n';
else
    std::cout << "Not found!\n";
```

> Found at position: 21

The search opportunities are further expanded by the following functions:

```
find_first_of     // Find first occurrence of characters
find_first_not_of // Find first absence of characters
find_last_of      // Find last occurrence of characters
find_last_not_of  // Find last absence of characters
```

These functions can allow you to search for characters from the end of the string, as well as find the negative case (ie. characters that are not in the string). Here is an example:

```
std::string str = "dog dog cat cat";
std::cout << "Found at position: " << str.find_last_of("gzx") << '\n';
```

> Found at position: 6

Note: Be aware that the above functions do not search for substrings, but rather for characters contained in the search string. In this case, the last occurrence of `'g'` was found at position `6` (the other characters weren't found).

Chapter 48: std::array

Parameter	Definition
`class T`	Specifies the data type of array members
`std::size_t N`	Specifies the number of members in the array

Section 48.1: Initializing an std::array

Initializing `std::array<T, N>`, where `T` is a scalar type and `N` is the number of elements of type `T`

If `T` is a scalar type, `std::array` can be initialized in the following ways:

```
// 1) Using aggregate-initialization
std::array<int, 3> a{ 0, 1, 2 };
// or equivalently
std::array<int, 3> a = { 0, 1, 2 };

// 2) Using the copy constructor
std::array<int, 3> a{ 0, 1, 2 };
std::array<int, 3> a2(a);
// or equivalently
std::array<int, 3> a2 = a;

// 3) Using the move constructor
std::array<int, 3> a = std::array<int, 3>{ 0, 1, 2 };
```

Initializing `std::array<T, N>`, where `T` is a non-scalar type and `N` is the number of elements of type `T`

If `T` is a non-scalar type `std::array` can be initialized in the following ways:

```
struct A { int values[3]; }; // An aggregate type

// 1) Using aggregate initialization with brace elision
// It works only if T is an aggregate type!
std::array<A, 2> a{ 0, 1, 2, 3, 4, 5 };
// or equivalently
std::array<A, 2> a = { 0, 1, 2, 3, 4, 5 };

// 2) Using aggregate initialization with brace initialization of sub-elements
std::array<A, 2> a{ A{ 0, 1, 2 }, A{ 3, 4, 5 } };
// or equivalently
std::array<A, 2> a = { A{ 0, 1, 2 }, A{ 3, 4, 5 } };

// 3)
std::array<A, 2> a{{ { 0, 1, 2 }, { 3, 4, 5 } }};
// or equivalently
std::array<A, 2> a = {{ { 0, 1, 2 }, { 3, 4, 5 } }};

// 4) Using the copy constructor
std::array<A, 2> a{ 1, 2, 3 };
std::array<A, 2> a2(a);
// or equivalently
std::array<A, 2> a2 = a;

// 5) Using the move constructor
std::array<A, 2> a = std::array<A, 2>{ 0, 1, 2, 3, 4, 5 };
```

Section 48.2: Element access

1. `at(pos)`

Returns a reference to the element at position `pos` with bounds checking. If `pos` is not within the range of the container, an exception of type `std::out_of_range` is thrown.

The complexity is constant O(1).

```
#include <array>

int main()
{
    std::array<int, 3> arr;

    // write values
    arr.at(0) = 2;
    arr.at(1) = 4;
    arr.at(2) = 6;

    // read values
    int a = arr.at(0); // a is now 2
    int b = arr.at(1); // b is now 4
    int c = arr.at(2); // c is now 6

    return 0;
}
```

2) `operator[pos]`

Returns a reference to the element at position `pos` without bounds checking. If `pos` is not within the range of the container, a runtime *segmentation violation* error can occur. This method provides element access equivalent to classic arrays and thereof more efficient than `at(pos)`.

The complexity is constant O(1).

```
#include <array>

int main()
{
    std::array<int, 3> arr;

    // write values
    arr[0] = 2;
    arr[1] = 4;
    arr[2] = 6;

    // read values
    int a = arr[0]; // a is now 2
    int b = arr[1]; // b is now 4
    int c = arr[2]; // c is now 6

    return 0;
}
```

3) `std::get<pos>`

This **non-member** function returns a reference to the element at **compile-time constant** position `pos` without

bounds checking. If `pos` is not within the range of the container, a runtime *segmentation violation* error can occur.

The complexity is constant O(1).

```
#include <array>

int main()
{
    std::array<int, 3> arr;

    // write values
    std::get<0>(arr) = 2;
    std::get<1>(arr) = 4;
    std::get<2>(arr) = 6;

    // read values
    int a = std::get<0>(arr); // a is now 2
    int b = std::get<1>(arr); // b is now 4
    int c = std::get<2>(arr); // c is now 6

    return 0;
}
```

4) `front()`

Returns a reference to the first element in container. Calling `front()` on an empty container is undefined.

The complexity is constant O(1).

Note: For a container c, the expression `c.front()` is equivalent to `*c.begin()`.

```
#include <array>

int main()
{
    std::array<int, 3> arr{ 2, 4, 6 };

    int a = arr.front(); // a is now 2

    return 0;
}
```

5) `back()`

Returns reference to the last element in the container. Calling `back()` on an empty container is undefined.

The complexity is constant O(1).

```
#include <array>

int main()
{
    std::array<int, 3> arr{ 2, 4, 6 };

    int a = arr.back(); // a is now 6

    return 0;
}
```

6) data()

Returns pointer to the underlying array serving as element storage. The pointer is such that `range [data(); data() + size())` is always a valid range, even if the container is empty (`data()` is not dereferenceable in that case).

The complexity is constant O(1).

```
#include <iostream>
#include <cstring>
#include <array>

int main ()
{
    const char* cstr = "Test string";
    std::array<char, 12> arr;

    std::memcpy(arr.data(), cstr, 12); // copy cstr to arr

    std::cout << arr.data(); // outputs: Test string

    return 0;
}
```

Section 48.3: Iterating through the Array

`std::array` being a STL container, can use range-based for loop similar to other containers like `vector`

```
int main() {
     std::array<int, 3> arr = { 1, 2, 3 };
     for (auto i : arr)
         cout << i << '\n';
}
```

Section 48.4: Checking size of the Array

One of the main advantage of `std::array` as compared to `C` style array is that we can check the size of the array using `size()` member function

```
int main() {
    std::array<int, 3> arr = { 1, 2, 3 };
    cout << arr.size() << endl;
}
```

Section 48.5: Changing all array elements at once

The member function `fill()` can be used on `std::array` for changing the values at once post initialization

```
int main() {

    std::array<int, 3> arr = { 1, 2, 3 };
    // change all elements of the array to 100
    arr.fill(100);

}
```

Chapter 49: std::vector

A vector is a dynamic array with automatically handled storage. The elements in a vector can be accessed just as efficiently as those in an array with the advantage being that vectors can dynamically change in size.

In terms of storage the vector data is (usually) placed in dynamically allocated memory thus requiring some minor overhead; conversely `C-arrays` and `std::array` use automatic storage relative to the declared location and thus do not have any overhead.

Section 49.1: Accessing Elements

There are two primary ways of accessing elements in a `std::vector`

- index-based access
- iterators

Index-based access:

This can be done either with the subscript operator `[]`, or the member function `at()`.

Both return a reference to the element at the respective position in the `std::vector` (unless it's a `vector<bool>`), so that it can be read as well as modified (if the vector is not `const`).

`[]` and `at()` differ in that `[]` is not guaranteed to perform any bounds checking, while `at()` does. Accessing elements where `index < 0` or `index >= size` is undefined behavior for `[]`, while `at()` throws a `std::out_of_range` exception.

Note: The examples below use C++11-style initialization for clarity, but the operators can be used with all versions (unless marked C++11).

Version ≥ C++11

```
std::vector<int> v{ 1, 2, 3 };
```

```
// using []
int a = v[1];    // a is 2
v[1] = 4;        // v now contains { 1, 4, 3 }

// using at()
int b = v.at(2); // b is 3
v.at(2) = 5;     // v now contains { 1, 4, 5 }
int c = v.at(3); // throws std::out_of_range exception
```

Because the `at()` method performs bounds checking and can throw exceptions, it is slower than `[]`. This makes `[]` preferred code where the semantics of the operation guarantee that the index is in bounds. In any case, accesses to elements of vectors are done in constant time. That means accessing to the first element of the vector has the same cost (in time) of accessing the second element, the third element and so on.

For example, consider this loop

```
for (std::size_t i = 0; i < v.size(); ++i) {
    v[i] = 1;
}
```

Here we know that the index variable `i` is always in bounds, so it would be a waste of CPU cycles to check that `i` is in bounds for every call to `operator[]`.

The front() and back() member functions allow easy reference access to the first and last element of the vector, respectively. These positions are frequently used, and the special accessors can be more readable than their alternatives using []:

```
std::vector<int> v{ 4, 5, 6 }; // In pre-C++11 this is more verbose

int a = v.front();   // a is 4, v.front() is equivalent to v[0]
v.front() = 3;       // v now contains {3, 5, 6}
int b = v.back();    // b is 6, v.back() is equivalent to v[v.size() - 1]
v.back() = 7;        // v now contains {3, 5, 7}
```

Note: It is undefined behavior to invoke `front()` or `back()` on an empty vector. You need to check that the container is not empty using the empty() member function (which checks if the container is empty) before calling `front()` or `back()`. A simple example of the use of 'empty()' to test for an empty vector follows:

```
int main ()
{
  std::vector<int> v;
  int sum (0);

  for (int i=1;i<=10;i++) v.push_back(i);//create and initialize the vector

  while (!v.empty())//loop through until the vector tests to be empty
  {
     sum += v.back();//keep a running total
     v.pop_back();//pop out the element which removes it from the vector
  }

  std::cout << "total: " << sum << '\n';//output the total to the user

  return 0;
}
```

The example above creates a vector with a sequence of numbers from 1 to 10. Then it pops the elements of the vector out until the vector is empty (using 'empty()') to prevent undefined behavior. Then the sum of the numbers in the vector is calculated and displayed to the user.

Version ≥ C++11

The data() method returns a pointer to the raw memory used by the `std::vector` to internally store its elements. This is most often used when passing the vector data to legacy code that expects a C-style array.

```
std::vector<int> v{ 1, 2, 3, 4 }; // v contains {1, 2, 3, 4}
int* p = v.data(); // p points to 1
*p = 4;            // v now contains {4, 2, 3, 4}
++p;               // p points to 2
*p = 3;            // v now contains {4, 3, 3, 4}
p[1] = 2;          // v now contains {4, 3, 2, 4}
*(p + 2) = 1;      // v now contains {4, 3, 2, 1}
```

Version < C++11

Before C++11, the `data()` method can be simulated by calling `front()` and taking the address of the returned value:

```
std::vector<int> v(4);
int* ptr = &(v.front()); // or &v[0]
```

This works because vectors are always guaranteed to store their elements in contiguous memory locations,

assuming the contents of the vector doesn't override unary `operator&`. If it does, you'll have to re-implement `std::addressof` in pre-C++11. It also assumes that the vector isn't empty.

Iterators:

Iterators are explained in more detail in the example "Iterating over `std::vector`" and the article Iterators. In short, they act similarly to pointers to the elements of the vector:

Version ≥ C++11

```
std::vector<int> v{ 4, 5, 6 };

auto it = v.begin();
int i = *it;        // i is 4
++it;
i = *it;            // i is 5
*it = 6;            // v contains { 4, 6, 6 }
auto e = v.end();   // e points to the element after the end of v. It can be
                    // used to check whether an iterator reached the end of the vector:
++it;
it == v.end();      // false, it points to the element at position 2 (with value 6)
++it;
it == v.end();      // true
```

It is consistent with the standard that a `std::vector<T>`'s iterators actually *be* `T*`s, but most standard libraries do not do this. Not doing this both improves error messages, catches non-portable code, and can be used to instrument the iterators with debugging checks in non-release builds. Then, in release builds, the class wrapping around the underlying pointer is optimized away.

You can persist a reference or a pointer to an element of a vector for indirect access. These references or pointers to elements in the `vector` remain stable and access remains defined unless you add/remove elements at or before the element in the `vector`, or you cause the `vector` capacity to change. This is the same as the rule for invalidating iterators.

Version ≥ C++11

```
std::vector<int> v{ 1, 2, 3 };
int* p = v.data() + 1;     // p points to 2
v.insert(v.begin(), 0);    // p is now invalid, accessing *p is a undefined behavior.
p = v.data() + 1;          // p points to 1
v.reserve(10);             // p is now invalid, accessing *p is a undefined behavior.
p = v.data() + 1;          // p points to 1
v.erase(v.begin());        // p is now invalid, accessing *p is a undefined behavior.
```

Section 49.2: Initializing a std::vector

A `std::vector` can be initialized in several ways while declaring it:

Version ≥ C++11

```
std::vector<int> v{ 1, 2, 3 };  // v becomes {1, 2, 3}

// Different from std::vector<int> v(3, 6)
std::vector<int> v{ 3, 6 };     // v becomes {3, 6}
```

```
// Different from std::vector<int> v{3, 6} in C++11
std::vector<int> v(3, 6);  // v becomes {6, 6, 6}

std::vector<int> v(4);     // v becomes {0, 0, 0, 0}
```

A vector can be initialized from another container in several ways:

Copy construction (from another vector only), which copies data from v2:

```
std::vector<int> v(v2);
std::vector<int> v = v2;
```

Version ≥ C++11

Move construction (from another vector only), which moves data from v2:

```
std::vector<int> v(std::move(v2));
std::vector<int> v = std::move(v2);
```

Iterator (range) copy-construction, which copies elements into v:

```
// from another vector
std::vector<int> v(v2.begin(), v2.begin() + 3); // v becomes {v2[0], v2[1], v2[2]}

// from an array
int z[] = { 1, 2, 3, 4 };
std::vector<int> v(z, z + 3);                   // v becomes {1, 2, 3}

// from a list
std::list<int> list1{ 1, 2, 3 };
std::vector<int> v(list1.begin(), list1.end()); // v becomes {1, 2, 3}
```

Version ≥ C++11

Iterator move-construction, using std::make_move_iterator, which moves elements into v:

```
// from another vector
std::vector<int> v(std::make_move_iterator(v2.begin()),
                   std::make_move_iterator(v2.end());

// from a list
std::list<int> list1{ 1, 2, 3 };
std::vector<int> v(std::make_move_iterator(list1.begin()),
                   std::make_move_iterator(list1.end()));
```

With the help of the assign() member function, a std::vector can be reinitialized after its construction:

```
v.assign(4, 100);                      // v becomes {100, 100, 100, 100}

v.assign(v2.begin(), v2.begin() + 3);  // v becomes {v2[0], v2[1], v2[2]}

int z[] = { 1, 2, 3, 4 };
v.assign(z + 1, z + 4);                // v becomes {2, 3, 4}
```

Section 49.3: Deleting Elements

Deleting the last element:

```
std::vector<int> v{ 1, 2, 3 };
v.pop_back();                         // v becomes {1, 2}
```

Deleting all elements:

```
std::vector<int> v{ 1, 2, 3 };
v.clear();                            // v becomes an empty vector
```

Deleting element by index:

```
std::vector<int> v{ 1, 2, 3, 4, 5, 6 };
v.erase(v.begin() + 3);               // v becomes {1, 2, 3, 5, 6}
```

Note: For a `vector` deleting an element which is not the last element, all elements beyond the deleted element have to be copied or moved to fill the gap, see the note below and std::list.

Deleting all elements in a range:

```
std::vector<int> v{ 1, 2, 3, 4, 5, 6 };
v.erase(v.begin() + 1, v.begin() + 5);  // v becomes {1, 6}
```

Note: The above methods do not change the capacity of the vector, only the size. See Vector Size and Capacity.

The `erase` method, which removes a range of elements, is often used as a part of the **erase-remove** idiom. That is, first `std::remove` moves some elements to the end of the vector, and then `erase` chops them off. This is a relatively inefficient operation for any indices less than the last index of the vector because all elements after the erased segments must be relocated to new positions. For speed critical applications that require efficient removal of arbitrary elements in a container, see std::list.

Deleting elements by value:

```
std::vector<int> v{ 1, 1, 2, 2, 3, 3 };
int value_to_remove = 2;
v.erase(std::remove(v.begin(), v.end(), value_to_remove), v.end()); // v becomes {1, 1, 3, 3}
```

Deleting elements by condition:

```
// std::remove_if needs a function, that takes a vector element as argument and returns true,
// if the element shall be removed
bool _predicate(const int& element) {
    return (element > 3); // This will cause all elements to be deleted that are larger than 3
}
...
std::vector<int> v{ 1, 2, 3, 4, 5, 6 };
v.erase(std::remove_if(v.begin(), v.end(), _predicate), v.end()); // v becomes {1, 2, 3}
```

Deleting elements by lambda, without creating additional predicate function

Version ≥ C++11

```
std::vector<int> v{ 1, 2, 3, 4, 5, 6 };
v.erase(std::remove_if(v.begin(), v.end(),
     [](auto& element){return element > 3;} ), v.end()
);
```

Deleting elements by condition from a loop:

```
std::vector<int> v{ 1, 2, 3, 4, 5, 6 };
std::vector<int>::iterator it = v.begin();
while (it != v.end()) {
    if (condition)
        it = v.erase(it); // after erasing, 'it' will be set to the next element in v
    else
        ++it;             // manually set 'it' to the next element in v
}
```

While it is important *not* to increment `it` in case of a deletion, you should consider using a different method when then erasing repeatedly in a loop. Consider `remove_if` for a more efficient way.

Deleting elements by condition from a reverse loop:

```
std::vector<int> v{ -1, 0, 1, 2, 3, 4, 5, 6 };
typedef std::vector<int>::reverse_iterator rev_itr;
rev_itr it = v.rbegin();

while (it != v.rend()) { // after the loop only '0' will be in v
    int value = *it;
    if (value) {
        ++it;
```

```
        // See explanation below for the following line.
        it = rev_itr(v.erase(it.base()));
    } else
        ++it;
}
```

Note some points for the preceding loop:

- Given a reverse iterator `it` pointing to some element, the method `base` gives the regular (non-reverse) iterator pointing to the same element.
- `vector::erase(iterator)` erases the element pointed to by an iterator, and returns an iterator to the element that followed the given element.
- `reverse_iterator::reverse_iterator(iterator)` constructs a reverse iterator from an iterator.

Put altogether, the line `it = rev_itr(v.erase(it.base()))` says: take the reverse iterator `it`, have `v` erase the element pointed by its regular iterator; take the resulting iterator, construct a reverse iterator from it, and assign it to the reverse iterator `it`.

Deleting all elements using `v.clear()` does not free up memory (`capacity()` of the vector remains unchanged). To reclaim space, use:

```
std::vector<int>().swap(v);
```

Version ≥ C++11

`shrink_to_fit()` frees up unused vector capacity:

```
v.shrink_to_fit();
```

The `shrink_to_fit` does not guarantee to really reclaim space, but most current implementations do.

Section 49.4: Iterating Over std::vector

You can iterate over a `std::vector` in several ways. For each of the following sections, `v` is defined as follows:

```
std::vector<int> v;
```

Iterating in the Forward Direction

Version ≥ C++11

```
// Range based for
for(const auto& value: v) {
    std::cout << value << "\n";
}

// Using a for loop with iterator
for(auto it = std::begin(v); it != std::end(v); ++it) {
    std::cout << *it << "\n";
}

// Using for_each algorithm, using a function or functor:
void fun(int const& value) {
    std::cout << value << "\n";
}

std::for_each(std::begin(v), std::end(v), fun);
```

```
// Using for_each algorithm. Using a lambda:
std::for_each(std::begin(v), std::end(v), [](int const& value) {
    std::cout << value << "\n";
});
```

Version < C++11

```
// Using a for loop with iterator
for(std::vector<int>::iterator it = std::begin(v); it != std::end(v); ++it) {
    std::cout << *it << "\n";
}
```

```
// Using a for loop with index
for(std::size_t i = 0; i < v.size(); ++i) {
    std::cout << v[i] << "\n";
}
```

Iterating in the Reverse Direction

Version ≥ C++14

```
// There is no standard way to use range based for for this.
// See below for alternatives.

// Using for_each algorithm
// Note: Using a lambda for clarity. But a function or functor will work
std::for_each(std::rbegin(v), std::rend(v), [](auto const& value) {
    std::cout << value << "\n";
});

// Using a for loop with iterator
for(auto rit = std::rbegin(v); rit != std::rend(v); ++rit) {
    std::cout << *rit << "\n";
}
```

```
// Using a for loop with index
for(std::size_t i = 0; i < v.size(); ++i) {
    std::cout << v[v.size() - 1 - i] << "\n";
}
```

Though there is no built-in way to use the range based for to reverse iterate; it is relatively simple to fix this. The range based for uses `begin()` and `end()` to get iterators and thus simulating this with a wrapper object can achieve the results we require.

Version ≥ C++14

```
template<class C>
struct ReverseRange {
  C c; // could be a reference or a copy, if the original was a temporary
  ReverseRange(C&& cin): c(std::forward<C>(cin)) {}
  ReverseRange(ReverseRange&&)=default;
  ReverseRange& operator=(ReverseRange&&)=delete;
  auto begin() const {return std::rbegin(c);}
  auto end()   const {return std::rend(c);}
};
// C is meant to be deduced, and perfect forwarded into
template<class C>
ReverseRange<C> make_ReverseRange(C&& c) {return {std::forward<C>(c)};}

int main() {
    std::vector<int> v { 1,2,3,4};
    for(auto const& value: make_ReverseRange(v)) {
        std::cout << value << "\n";
    }
}
```

Enforcing const elements

Since C++11 the `cbegin()` and `cend()` methods allow you to obtain a *constant iterator* for a vector, even if the vector is non-const. A constant iterator allows you to read but not modify the contents of the vector which is useful to enforce const correctness:

Version ≥ C++11

```
// forward iteration
for (auto pos = v.cbegin(); pos != v.cend(); ++pos) {
   // type of pos is vector<T>::const_iterator
   // *pos = 5; // Compile error - can't write via const iterator
}

// reverse iteration
for (auto pos = v.crbegin(); pos != v.crend(); ++pos) {
   // type of pos is vector<T>::const_iterator
   // *pos = 5; // Compile error - can't write via const iterator
}

// expects Functor::operand()(T&)
for_each(v.begin(), v.end(), Functor());

// expects Functor::operand()(const T&)
for_each(v.cbegin(), v.cend(), Functor())
```

Version ≥ C++17

as_const extends this to range iteration:

```
for (auto const& e : std::as_const(v)) {
  std::cout << e << '\n';
}
```

This is easy to implement in earlier versions of C++:

Version ≥ C++14

```
template <class T>
constexpr std::add_const_t<T>& as_const(T& t) noexcept {
  return t;
}
```

A Note on Efficiency

Since the class `std::vector` is basically a class that manages a dynamically allocated contiguous array, the same principle explained here applies to C++ vectors. Accessing the vector's content by index is much more efficient when following the row-major order principle. Of course, each access to the vector also puts its management content into the cache as well, but as has been debated many times (notably here and here), the difference in performance for iterating over a `std::vector` compared to a raw array is negligible. So the same principle of efficiency for raw arrays in C also applies for C++'s `std::vector`.

Section 49.5: vector<bool>: The Exception To So Many, So Many Rules

The standard (section 23.3.7) specifies that a specialization of `vector<bool>` is provided, which optimizes space by packing the `bool` values, so that each takes up only one bit. Since bits aren't addressable in C++, this means that several requirements on `vector` are not placed on `vector<bool>`:

- The data stored is not required to be contiguous, so a `vector<bool>` can't be passed to a C API which expects a `bool` array.
- `at()`, `operator []`, and dereferencing of iterators do not return a reference to `bool`. Rather they return a

proxy object that (imperfectly) simulates a reference to a `bool` by overloading its assignment operators. As an example, the following code may not be valid for `std::vector<bool>`, because dereferencing an iterator does not return a reference:

Version ≥ C++11

```
std::vector<bool> v = {true, false};
for (auto &b: v) { } // error
```

Similarly, functions expecting a `bool&` argument cannot be used with the result of `operator []` or `at()` applied to `vector<bool>`, or with the result of dereferencing its iterator:

```
  void f(bool& b);
  f(v[0]);             // error
  f(*v.begin());       // error
```

The implementation of `std::vector<bool>` is dependent on both the compiler and architecture. The specialisation is implemented by packing `n` Booleans into the lowest addressable section of memory. Here, `n` is the size in bits of the lowest addressable memory. In most modern systems this is 1 byte or 8 bits. This means that one byte can store 8 Boolean values. This is an improvement over the traditional implementation where 1 Boolean value is stored in 1 byte of memory.

Note: The below example shows possible bitwise values of individual bytes in a traditional vs. optimized `vector<bool>`. This will not always hold true in all architectures. It is, however, a good way of visualising the optimization. In the below examples a byte is represented as [x, x, x, x, x, x, x, x].

Traditional *`std::vector<char>` storing 8 Boolean values:*

Version ≥ C++11

```
std::vector<char> trad_vect = {true, false, false, false, true, false, true, true};
```

Bitwise representation:

```
[0,0,0,0,0,0,0,1], [0,0,0,0,0,0,0,0], [0,0,0,0,0,0,0,0], [0,0,0,0,0,0,0,0],
[0,0,0,0,0,0,0,1], [0,0,0,0,0,0,0,0], [0,0,0,0,0,0,0,1], [0,0,0,0,0,0,0,1]
```

Specialized *`std::vector<bool>` storing 8 Boolean values:*

Version ≥ C++11

```
std::vector<bool> optimized_vect = {true, false, false, false, true, false, true, true};
```

Bitwise representation:

```
[1,0,0,0,1,0,1,1]
```

Notice in the above example, that in the traditional version of `std::vector<bool>`, 8 Boolean values take up 8 bytes of memory, whereas in the optimized version of `std::vector<bool>`, they only use 1 byte of memory. This is a significant improvement on memory usage. If you need to pass a `vector<bool>` to an C-style API, you may need to copy the values to an array, or find a better way to use the API, if memory and performance are at risk.

Section 49.6: Inserting Elements

Appending an element at the end of a vector (by copying/moving):

```
struct Point {
  double x, y;
```

```
  Point(double x, double y) : x(x), y(y) {}
};
std::vector<Point> v;
Point p(10.0, 2.0);
v.push_back(p);  // p is copied into the vector.
```

Version ≥ C++11

Appending an element at the end of a vector by constructing the element in place:

```
std::vector<Point> v;
v.emplace_back(10.0, 2.0); // The arguments are passed to the constructor of the
                           // given type (here Point). The object is constructed
                           // in the vector, avoiding a copy.
```

Note that `std::vector` does *not* have a `push_front()` member function due to performance reasons. Adding an element at the beginning causes all existing elements in the vector to be moved. If you want to frequently insert elements at the beginning of your container, then you might want to use `std::list` or `std::deque` instead.

Inserting an element at any position of a vector:

```
std::vector<int> v{ 1, 2, 3 };
v.insert(v.begin(), 9);          // v now contains {9, 1, 2, 3}
```

Version ≥ C++11

Inserting an element at any position of a vector by constructing the element in place:

```
std::vector<int> v{ 1, 2, 3 };
v.emplace(v.begin()+1, 9);     // v now contains {1, 9, 2, 3}
```

Inserting another vector at any position of the vector:

```
std::vector<int> v(4);      // contains: 0, 0, 0, 0
std::vector<int> v2(2, 10); // contains: 10, 10
v.insert(v.begin()+2, v2.begin(), v2.end()); // contains: 0, 0, 10, 10, 0, 0
```

Inserting an array at any position of a vector:

```
std::vector<int> v(4); // contains: 0, 0, 0, 0
int a [] = {1, 2, 3}; // contains: 1, 2, 3
v.insert(v.begin()+1, a, a+sizeof(a)/sizeof(a[0])); // contains: 0, 1, 2, 3, 0, 0, 0
```

Use `reserve()` before inserting multiple elements if resulting vector size is known beforehand to avoid multiple reallocations (see vector size and capacity):

```
std::vector<int> v;
v.reserve(100);
for(int i = 0; i < 100; ++i)
    v.emplace_back(i);
```

Be sure to not make the mistake of calling `resize()` in this case, or you will inadvertently create a vector with 200 elements where only the latter one hundred will have the value you intended.

Section 49.7: Using std::vector as a C array

There are several ways to use a `std::vector` as a C array (for example, for compatibility with C libraries). This is possible because the elements in a vector are stored contiguously.

Version ≥ C++11

```
std::vector<int> v{ 1, 2, 3 };
int* p = v.data();
```

In contrast to solutions based on previous C++ standards (see below), the member function `.data()` may also be applied to empty vectors, because it doesn't cause undefined behavior in this case.

Before C++11, you would take the address of the vector's first element to get an equivalent pointer, if the vector isn't empty, these both methods are interchangeable:

```
int* p = &v[0];      // combine subscript operator and 0 literal

int* p = &v.front(); // explicitly reference the first element
```

Note: If the vector is empty, `v[0]` and `v.front()` are undefined and cannot be used.

When storing the base address of the vector's data, note that many operations (such as `push_back`, `resize`, etc.) can change the data memory location of the vector, thus invalidating previous data pointers. For example:

```
std::vector<int> v;
int* p = v.data();
v.resize(42);      // internal memory location changed; value of p is now invalid
```

Section 49.8: Finding an Element in std::vector

The function `std::find`, defined in the **`<algorithm>`** header, can be used to find an element in a `std::vector`.

`std::find` uses the `operator==` to compare elements for equality. It returns an iterator to the first element in the range that compares equal to the value.

If the element in question is not found, `std::find` returns `std::vector::end` (or `std::vector::cend` if the vector is `const`).

Version < C++11

```
static const int arr[] = {5, 4, 3, 2, 1};
std::vector<int> v (arr, arr + sizeof(arr) / sizeof(arr[0]) );

std::vector<int>::iterator it = std::find(v.begin(), v.end(), 4);
std::vector<int>::difference_type index = std::distance(v.begin(), it);
// `it` points to the second element of the vector, `index` is 1

std::vector<int>::iterator missing = std::find(v.begin(), v.end(), 10);
std::vector<int>::difference_type index_missing = std::distance(v.begin(), missing);
// `missing` is v.end(), `index_missing` is 5 (ie. size of the vector)
```

Version ≥ C++11

```
std::vector<int> v { 5, 4, 3, 2, 1 };

auto it = std::find(v.begin(), v.end(), 4);
auto index = std::distance(v.begin(), it);
// `it` points to the second element of the vector, `index` is 1

auto missing = std::find(v.begin(), v.end(), 10);
auto index_missing = std::distance(v.begin(), missing);
// `missing` is v.end(), `index_missing` is 5 (ie. size of the vector)
```

If you need to perform many searches in a large vector, then you may want to consider sorting the vector first,

before using the binary_search algorithm.

To find the first element in a vector that satisfies a condition, `std::find_if` can be used. In addition to the two parameters given to `std::find`, `std::find_if` accepts a third argument which is a function object or function pointer to a predicate function. The predicate should accept an element from the container as an argument and return a value convertible to `bool`, without modifying the container:

Version < C++11

```
bool isEven(int val) {
    return (val % 2 == 0);
}

struct moreThan {
    moreThan(int limit) : _limit(limit) {}

    bool operator()(int val) {
        return val > _limit;
    }

    int _limit;
};

static const int arr[] = {1, 3, 7, 8};
std::vector<int> v (arr, arr + sizeof(arr) / sizeof(arr[0]) );

std::vector<int>::iterator it = std::find_if(v.begin(), v.end(), isEven);
// `it` points to 8, the first even element

std::vector<int>::iterator missing = std::find_if(v.begin(), v.end(), moreThan(10));
// `missing` is v.end(), as no element is greater than 10
```

Version ≥ C++11

```
// find the first value that is even
std::vector<int> v = {1, 3, 7, 8};
auto it = std::find_if(v.begin(), v.end(), [](int val){return val % 2 == 0;});
// `it` points to 8, the first even element

auto missing = std::find_if(v.begin(), v.end(), [](int val){return val > 10;});
// `missing` is v.end(), as no element is greater than 10
```

Section 49.9: Concatenating Vectors

One `std::vector` can be append to another by using the member function `insert()`:

```
std::vector<int> a = {0, 1, 2, 3, 4};
std::vector<int> b = {5, 6, 7, 8, 9};

a.insert(a.end(), b.begin(), b.end());
```

However, this solution fails if you try to append a vector to itself, because the standard specifies that iterators given to `insert()` must not be from the same range as the receiver object's elements.

Version ≥ c++11

Instead of using the vector's member functions, the functions `std::begin()` and `std::end()` can be used:

```
a.insert(std::end(a), std::begin(b), std::end(b));
```

This is a more general solution, for example, because `b` can also be an array. However, also this solution doesn't

allow you to append a vector to itself.

If the order of the elements in the receiving vector doesn't matter, considering the number of elements in each vector can avoid unnecessary copy operations:

```
if (b.size() < a.size())
  a.insert(a.end(), b.begin(), b.end());
else
  b.insert(b.end(), a.begin(), a.end());
```

Section 49.10: Matrices Using Vectors

Vectors can be used as a 2D matrix by defining them as a vector of vectors.

A matrix with 3 rows and 4 columns with each cell initialised as 0 can be defined as:

```
std::vector<std::vector<int> > matrix(3, std::vector<int>(4));
```

Version ≥ C++11

The syntax for initializing them using initialiser lists or otherwise are similar to that of a normal vector.

```
  std::vector<std::vector<int>> matrix = { {0,1,2,3},
                                           {4,5,6,7},
                                           {8,9,10,11}
                                         };
```

Values in such a vector can be accessed similar to a 2D array

```
int var = matrix[0][2];
```

Iterating over the entire matrix is similar to that of a normal vector but with an extra dimension.

```
for(int i = 0; i < 3; ++i)
{
    for(int j = 0; j < 4; ++j)
    {
        std::cout << matrix[i][j] << std::endl;
    }
}
```

Version ≥ C++11

```
for(auto& row: matrix)
{
    for(auto& col : row)
    {
        std::cout << col << std::endl;
    }
}
```

A vector of vectors is a convenient way to represent a matrix but it's not the most efficient: individual vectors are scattered around memory and the data structure isn't cache friendly.

Also, in a proper matrix, the length of every row must be the same (this isn't the case for a vector of vectors). The additional flexibility can be a source of errors.

Section 49.11: Using a Sorted Vector for Fast Element Lookup

The `<algorithm>` header provides a number of useful functions for working with sorted vectors.

An important prerequisite for working with sorted vectors is that the stored values are comparable with `<`.

An unsorted vector can be sorted by using the function `std::sort()`:

```
std::vector<int> v;
// add some code here to fill v with some elements
std::sort(v.begin(), v.end());
```

Sorted vectors allow efficient element lookup using the function `std::lower_bound()`. Unlike `std::find()`, this performs an efficient binary search on the vector. The downside is that it only gives valid results for sorted input ranges:

```
// search the vector for the first element with value 42
std::vector<int>::iterator it = std::lower_bound(v.begin(), v.end(), 42);
if (it != v.end() && *it == 42) {
    // we found the element!
}
```

Note: If the requested value is not part of the vector, `std::lower_bound()` will return an iterator to the first element that is *greater* than the requested value. This behavior allows us to insert a new element at its right place in an already sorted vector:

```
int const new_element = 33;
v.insert(std::lower_bound(v.begin(), v.end(), new_element), new_element);
```

If you need to insert a lot of elements at once, it might be more efficient to call `push_back()` for all them first and then call `std::sort()` once all elements have been inserted. In this case, the increased cost of the sorting can pay off against the reduced cost of inserting new elements at the end of the vector and not in the middle.

If your vector contains multiple elements of the same value, `std::lower_bound()` will try to return an iterator to the first element of the searched value. However, if you need to insert a new element *after* the last element of the searched value, you should use the function `std::upper_bound()` as this will cause less shifting around of elements:

```
v.insert(std::upper_bound(v.begin(), v.end(), new_element), new_element);
```

If you need both the upper bound and the lower bound iterators, you can use the function `std::equal_range()` to retrieve both of them efficiently with one call:

```
std::pair<std::vector<int>::iterator,
          std::vector<int>::iterator> rg = std::equal_range(v.begin(), v.end(), 42);
std::vector<int>::iterator lower_bound = rg.first;
std::vector<int>::iterator upper_bound = rg.second;
```

In order to test whether an element exists in a sorted vector (although not specific to vectors), you can use the function `std::binary_search()`:

```
bool exists = std::binary_search(v.begin(), v.end(), value_to_find);
```

Section 49.12: Reducing the Capacity of a Vector

A `std::vector` automatically increases its capacity upon insertion as needed, but it never reduces its capacity after element removal.

```
// Initialize a vector with 100 elements
std::vector<int> v(100);

// The vector's capacity is always at least as large as its size
auto const old_capacity = v.capacity();
// old_capacity >= 100

// Remove half of the elements
v.erase(v.begin() + 50, v.end());  // Reduces the size from 100 to 50 (v.size() == 50),
                                   // but not the capacity (v.capacity() == old_capacity)
```

To reduce its capacity, we can copy the contents of a vector to a new temporary vector. The new vector will have the minimum capacity that is needed to store all elements of the original vector. If the size reduction of the original vector was significant, then the capacity reduction for the new vector is likely to be significant. We can then swap the original vector with the temporary one to retain its minimized capacity:

```
std::vector<int>(v).swap(v);
```

Version ≥ C++11

In C++11 we can use the `shrink_to_fit()` member function for a similar effect:

```
v.shrink_to_fit();
```

Note: The `shrink_to_fit()` member function is a request and doesn't guarantee to reduce capacity.

Section 49.13: Vector size and capacity

Vector size is simply the number of elements in the vector:

1. Current vector **size** is queried by `size()` member function. Convenience `empty()` function returns `true` if size is 0:

```
vector<int> v = { 1, 2, 3 }; // size is 3
const vector<int>::size_type size = v.size();
cout << size << endl; // prints 3
cout << boolalpha << v.empty() << endl; // prints false
```

2. Default constructed vector starts with a size of 0:

```
vector<int> v; // size is 0
cout << v.size() << endl; // prints 0
```

3. Adding `N` elements to vector increases **size** by `N` (e.g. by `push_back()`, `insert()` or `resize()` functions).

4. Removing `N` elements from vector decreases **size** by `N` (e.g. by `pop_back()`, `erase()` or `clear()` functions).

5. Vector has an implementation-specific upper limit on its size, but you are likely to run out of RAM before reaching it:

```
vector<int> v;
```

```
const vector<int>::size_type max_size = v.max_size();
cout << max_size << endl; // prints some large number
v.resize( max_size ); // probably won't work
v.push_back( 1 ); // definitely won't work
```

Common mistake: **size** is not necessarily (or even usually) `int`:

```
// !!!bad!!!evil!!!
vector<int> v_bad( N, 1 ); // constructs large N size vector
for( int i = 0; i < v_bad.size(); ++i ) { // size is not supposed to be int!
    do_something( v_bad[i] );
}
```

Vector capacity differs from **size**. While **size** is simply how many elements the vector currently has, **capacity** is for how many elements it allocated/reserved memory for. That is useful, because too frequent (re)allocations of too large sizes can be expensive.

1. Current vector **capacity** is queried by `capacity()` member function. **Capacity** is always greater or equal to **size**:

   ```
   vector<int> v = { 1, 2, 3 }; // size is 3, capacity is >= 3
   const vector<int>::size_type capacity = v.capacity();
   cout << capacity << endl; // prints number >= 3
   ```

2. You can manually reserve capacity by `reserve( N )` function (it changes vector capacity to `N`):

   ```
   // !!!bad!!!evil!!!
   vector<int> v_bad;
   for( int i = 0; i < 10000; ++i ) {
       v_bad.push_back( i ); // possibly lot of reallocations
   }

   // good
   vector<int> v_good;
   v_good.reserve( 10000 ); // good! only one allocation
   for( int i = 0; i < 10000; ++i ) {
       v_good.push_back( i ); // no allocations needed anymore
   }
   ```

3. You can request for the excess capacity to be released by `shrink_to_fit()` (but the implementation doesn't have to obey you). This is useful to conserve used memory:

   ```
   vector<int> v = { 1, 2, 3, 4, 5 }; // size is 5, assume capacity is 6
   v.shrink_to_fit(); // capacity is 5 (or possibly still 6)
   cout << boolalpha << v.capacity() == v.size() << endl; // prints likely true (but possibly
   false)
   ```

Vector partly manages capacity automatically, when you add elements it may decide to grow. Implementers like to use 2 or 1.5 for the grow factor (golden ratio would be the ideal value - but is impractical due to being rational number). On the other hand vector usually do not automatically shrink. For example:

```
vector<int> v; // capacity is possibly (but not guaranteed) to be 0
v.push_back( 1 ); // capacity is some starter value, likely 1
v.clear(); // size is 0 but capacity is still same as before!
```

```
v = { 1, 2, 3, 4 }; // size is 4, and lets assume capacity is 4.
v.push_back( 5 ); // capacity grows - let's assume it grows to 6 (1.5 factor)
v.push_back( 6 ); // no change in capacity
v.push_back( 7 ); // capacity grows - let's assume it grows to 9 (1.5 factor)
// and so on
v.pop_back(); v.pop_back(); v.pop_back(); v.pop_back(); // capacity stays the same
```

Section 49.14: Iterator/Pointer Invalidation

Iterators and pointers pointing into an `std::vector` can become invalid, but only when performing certain operations. Using invalid iterators/pointers will result in undefined behavior.

Operations which invalidate iterators/pointers include:

- Any insertion operation which changes the `capacity` of the `vector` will invalidate *all* iterators/pointers:

```
vector<int> v(5); // Vector has a size of 5; capacity is unknown.
int *p1 = &v[0];
v.push_back(2);   // p1 may have been invalidated, since the capacity was unknown.

v.reserve(20);    // Capacity is now at least 20.
int *p2 = &v[0];
v.push_back(4);   // p2 is *not* invalidated, since the size of `v` is now 7.
v.insert(v.end(), 30, 9); // Inserts 30 elements at the end. The size exceeds the
                          // requested capacity of 20, so `p2` is (probably) invalidated.
int *p3 = &v[0];
v.reserve(v.capacity() + 20); // Capacity exceeded, thus `p3` is invalid.
```

Version ≥ C++11

```
auto old_cap = v.capacity();
v.shrink_to_fit();
if(old_cap != v.capacity())
    // Iterators were invalidated.
```

- Any insertion operation, which does not increase the capacity, will still invalidate iterators/pointers pointing to elements at the insertion position and past it. This includes the `end` iterator:

```
vector<int> v(5);
v.reserve(20);                 // Capacity is at least 20.
int *p1 = &v[0];
int *p2 = &v[3];
v.insert(v.begin() + 2, 5, 0); // `p2` is invalidated, but since the capacity
                               // did not change, `p1` remains valid.
int *p3 = &v[v.size() - 1];
v.push_back(10); // The capacity did not change, so `p3` and `p1` remain valid.
```

- Any removal operation will invalidate iterators/pointers pointing to the removed elements and to any elements past the removed elements. This includes the `end` iterator:

```
vector<int> v(10);
int *p1 = &v[0];
int *p2 = &v[5];
v.erase(v.begin() + 3, v.end()); // `p2` is invalid, but `p1` remains valid.
```

- `operator=` (copy, move, or otherwise) and `clear()` will invalidate all iterators/pointers pointing into the vector.

Section 49.15: Find max and min Element and Respective Index in a Vector

To find the largest or smallest element stored in a vector, you can use the methods `std::max_element` and `std::min_element`, respectively. These methods are defined in **<algorithm>** header. If several elements are equivalent to the greatest (smallest) element, the methods return the iterator to the first such element. Return `v.end()` for empty vectors.

```
std::vector<int> v = {5, 2, 8, 10, 9};
int maxElementIndex = std::max_element(v.begin(),v.end()) - v.begin();
int maxElement = *std::max_element(v.begin(), v.end());

int minElementIndex = std::min_element(v.begin(),v.end()) - v.begin();
int minElement = *std::min_element(v.begin(), v.end());

std::cout << "maxElementIndex:" << maxElementIndex << ", maxElement:" << maxElement << '\n';
std::cout << "minElementIndex:" << minElementIndex << ", minElement:" << minElement << '\n';
```

Output:

> maxElementIndex:3, maxElement:10
> minElementIndex:1, minElement:2

Version ≥ C++11

The minimum and maximum element in a vector can be retrieved at the same time by using the method `std::minmax_element`, which is also defined in **<algorithm>** header:

```
std::vector<int> v = {5, 2, 8, 10, 9};
auto minmax = std::minmax_element(v.begin(), v.end());

std::cout << "minimum element: " << *minmax.first << '\n';
std::cout << "maximum element: " << *minmax.second << '\n';
```

Output:

> minimum element: 2
> maximum element: 10

Section 49.16: Converting an array to std::vector

An array can easily be converted into a `std::vector` by using `std::begin` and `std::end`:

Version ≥ C++11

```
int values[5] = { 1, 2, 3, 4, 5 }; // source array

std::vector<int> v(std::begin(values), std::end(values)); // copy array to new vector

for(auto &x: v)
    std::cout << x << " ";
std::cout << std::endl;
```

> 1 2 3 4 5

```
int main(int argc, char* argv[]) {
    // convert main arguments into a vector of strings.
    std::vector<std::string>  args(argv, argv + argc);
}
```

A C++11 initializer_list<> can also be used to initialize the vector at once

```
initializer_list<int> arr = { 1,2,3,4,5 };
vector<int> vec1 {arr};

for (auto & i : vec1)
    cout << i << endl;
```

Section 49.17: Functions Returning Large Vectors

Version ≥ C++11

In C++11, compilers are required to implicitly move from a local variable that is being returned. Moreover, most compilers can perform copy elision in many cases and elide the move altogether. As a result of this, returning large objects that can be moved cheaply no longer requires special handling:

```
#include <vector>
#include <iostream>

// If the compiler is unable to perform named return value optimization (NRVO)
// and elide the move altogether, it is required to move from v into the return value.
std::vector<int> fillVector(int a, int b) {
    std::vector<int> v;
    v.reserve(b-a+1);
    for (int i = a; i <= b; i++) {
        v.push_back(i);
    }
    return v; // implicit move
}

int main() { // declare and fill vector
    std::vector<int> vec = fillVector(1, 10);

    // print vector
    for (auto value : vec)
        std::cout << value << " "; // this will print "1 2 3 4 5 6 7 8 9 10 "

    std::cout << std::endl;

    return 0;
}
```

Version < C++11

Before C++11, copy elision was already allowed and implemented by most compilers. However, due to the absence of move semantics, in legacy code or code that has to be compiled with older compiler versions which don't implement this optimization, you can find vectors being passed as output arguments to prevent the unneeded copy:

```
#include <vector>
#include <iostream>
```

```
// passing a std::vector by reference
void fillVectorFrom_By_Ref(int a, int b, std::vector<int> &v) {
    assert(v.empty());
    v.reserve(b-a+1);
    for (int i = a; i <= b; i++) {
        v.push_back(i);
    }
}

int main() {// declare vector
    std::vector<int> vec;

    // fill vector
    fillVectorFrom_By_Ref(1, 10, vec);
    // print vector
    for (std::vector<int>::const_iterator it = vec.begin(); it != vec.end(); ++it)
        std::cout << *it << " "; // this will print "1 2 3 4 5 6 7 8 9 10 "
    std::cout << std::endl;
    return 0;
}
```

Chapter 50: std::map

- To use any of `std::map` or `std::multimap` the header file **`<map>`** should be included.
- `std::map` and `std::multimap` both keep their elements sorted according to the ascending order of keys. In case of `std::multimap`, no sorting occurs for the values of the same key.
- The basic difference between `std::map` and `std::multimap` is that the `std::map` one does not allow duplicate values for the same key where `std::multimap` does.
- Maps are implemented as binary search trees. So `search()`, `insert()`, `erase()` takes Θ(log n) time in average. For constant time operation use `std::unordered_map`.
- `size()` and `empty()` functions have Θ(1) time complexity, number of nodes is cached to avoid walking through tree each time these functions are called.

Section 50.1: Accessing elements

An `std::map` takes `(key, value)` pairs as input.

Consider the following example of `std::map` initialization:

```
std::map < std::string, int > ranking { std::make_pair("stackoverflow", 2),
                                         std::make_pair("docs-beta", 1) };
```

In an `std::map` , elements can be inserted as follows:

```
ranking["stackoverflow"]=2;
ranking["docs-beta"]=1;
```

In the above example, if the key `stackoverflow` is already present, its value will be updated to 2. If it isn't already present, a new entry will be created.

In an `std::map`, elements can be accessed directly by giving the key as an index:

```
std::cout << ranking[ "stackoverflow" ] << std::endl;
```

Note that using the `operator[]` on the map will actually *insert a new value* with the queried key into the map. This means that you cannot use it on a `const std::map`, even if the key is already stored in the map. To prevent this insertion, check if the element exists (for example by using `find()`) or use `at()` as described below.

Version ≥ C++11

Elements of a `std::map` can be accessed with `at()`:

```
std::cout << ranking.at("stackoverflow") << std::endl;
```

Note that `at()` will throw an `std::out_of_range` exception if the container does not contain the requested element.

In both containers `std::map` and `std::multimap`, elements can be accessed using iterators:

Version ≥ C++11

```
// Example using begin()
```

```
std::multimap < int, std::string > mmp { std::make_pair(2, "stackoverflow"),
                                         std::make_pair(1, "docs-beta"),
                                         std::make_pair(2, "stackexchange")  };
auto it = mmp.begin();
std::cout << it->first << " : " << it->second << std::endl; // Output: "1 : docs-beta"
it++;
std::cout << it->first << " : " << it->second << std::endl; // Output: "2 : stackoverflow"
it++;
std::cout << it->first << " : " << it->second << std::endl; // Output: "2 : stackexchange"

// Example using rbegin()
std::map < int, std::string > mp {  std::make_pair(2, "stackoverflow"),
                                    std::make_pair(1, "docs-beta"),
                                    std::make_pair(2, "stackexchange")  };
auto it2 = mp.rbegin();
std::cout << it2->first << " : " << it2->second << std::endl; // Output: "2 : stackoverflow"
it2++;
std::cout << it2->first << " : " << it2->second << std::endl; // Output: "1 : docs-beta"
```

Section 50.2: Inserting elements

An element can be inserted into a `std::map` only if its key is not already present in the map. Given for example:

```
std::map< std::string, size_t > fruits_count;
```

- A key-value pair is inserted into a `std::map` through the `insert()` member function. It requires a `pair` as an argument:

  ```
  fruits_count.insert({"grapes", 20});
  fruits_count.insert(make_pair("orange", 30));
  fruits_count.insert(pair<std::string, size_t>("banana", 40));
  fruits_count.insert(map<std::string, size_t>::value_type("cherry", 50));
  ```

  The `insert()` function returns a `pair` consisting of an iterator and a `bool` value:

  - If the insertion was successful, the iterator points to the newly inserted element, and the `bool` value is `true`.
  - If there was already an element with the same `key`, the insertion fails. When that happens, the iterator points to the element causing the conflict, and the `bool` is value is `false`.

  The following method can be used to combine insertion and searching operation:

  ```
  auto success = fruits_count.insert({"grapes", 20});
  if (!success.second) {           // we already have 'grapes' in the map
      success.first->second += 20; // access the iterator to update the value
  }
  ```

- For convenience, the `std::map` container provides the subscript operator to access elements in the map and to insert new ones if they don't exist:

  ```
  fruits_count["apple"] = 10;
  ```

  While simpler, it prevents the user from checking if the element already exists. If an element is missing, `std::map::operator[]` implicitly creates it, initializing it with the default constructor before overwriting it with the supplied value.

- `insert()` can be used to add several elements at once using a braced list of pairs. This version of insert() returns void:

```
fruits_count.insert({{"apricot", 1}, {"jackfruit", 1}, {"lime", 1}, {"mango", 7}});
```

- `insert()` can also be used to add elements by using iterators denoting the begin and end of `value_type` values:

```
std::map< std::string, size_t > fruit_list{ {"lemon", 0}, {"olive", 0}, {"plum", 0}};
fruits_count.insert(fruit_list.begin(), fruit_list.end());
```

Example:

```
std::map<std::string, size_t> fruits_count;
std::string fruit;
while(std::cin >> fruit){
    // insert an element with 'fruit' as key and '1' as value
    // (if the key is already stored in fruits_count, insert does nothing)
    auto ret = fruits_count.insert({fruit, 1});
    if(!ret.second){            // 'fruit' is already in the map
        ++ret.first->second;    // increment the counter
    }
}
```

Time complexity for an insertion operation is O(log n) because `std::map` are implemented as trees.

Version ≥ C++11

A `pair` can be constructed explicitly using `make_pair()` and `emplace()`:

```
std::map< std::string , int > runs;
runs.emplace("Babe Ruth", 714);
runs.insert(make_pair("Barry Bonds", 762));
```

If we know where the new element will be inserted, then we can use `emplace_hint()` to specify an iterator `hint`. If the new element can be inserted just before `hint`, then the insertion can be done in constant time. Otherwise it behaves in the same way as `emplace()`:

```
std::map< std::string , int > runs;
auto it = runs.emplace("Barry Bonds", 762); // get iterator to the inserted element
// the next element will be before "Barry Bonds", so it is inserted before 'it'
runs.emplace_hint(it, "Babe Ruth", 714);
```

Section 50.3: Searching in std::map or in std::multimap

There are several ways to search a key in `std::map` or in `std::multimap`.

- To get the iterator of the first occurrence of a key, the `find()` function can be used. It returns `end()` if the key does not exist.

```
std::multimap< int , int > mmp{ {1, 2}, {3, 4}, {6, 5}, {8, 9}, {3, 4}, {6, 7} };
auto it = mmp.find(6);
if(it!=mmp.end())
    std::cout << it->first << ", " << it->second << std::endl; //prints: 6, 5
else
```

```
    std::cout << "Value does not exist!" << std::endl;

it = mmp.find(66);
if(it!=mmp.end())
    std::cout << it->first << ", " << it->second << std::endl;
else
    std::cout << "Value does not exist!" << std::endl; // This line would be executed.
```

- Another way to find whether an entry exists in `std::map` or in `std::multimap` is using the `count()` function, which counts how many values are associated with a given key. Since `std::map` associates only one value with each key, its `count()` function can only return 0 (if the key is not present) or 1 (if it is). For `std::multimap`, `count()` can return values greater than 1 since there can be several values associated with the same key.

```
std::map< int , int > mp{ {1, 2}, {3, 4}, {6, 5}, {8, 9}, {3, 4}, {6, 7} };
if(mp.count(3) > 0) // 3 exists as a key in map
    std::cout << "The key exists!" << std::endl; // This line would be executed.
else
    std::cout << "The key does not exist!" << std::endl;
```

If you only care whether some element exists, `find` is strictly better: it documents your intent and, for `multimaps`, it can stop once the first matching element has been found.

- In the case of `std::multimap`, there could be several elements having the same key. To get this range, the `equal_range()` function is used which returns `std::pair` having iterator lower bound (inclusive) and upper bound (exclusive) respectively. If the key does not exist, both iterators would point to `end()`.

```
auto eqr = mmp.equal_range(6);
auto st = eqr.first, en = eqr.second;
for(auto it = st; it != en; ++it){
    std::cout << it->first << ", " << it->second << std::endl;
}
    // prints: 6, 5
    //         6, 7
```

Section 50.4: Initializing a std::map or std::multimap

`std::map` and `std::multimap` both can be initialized by providing key-value pairs separated by comma. Key-value pairs could be provided by either `{key, value}` or can be explicitly created by `std::make_pair(key, value)`. As `std::map` does not allow duplicate keys and comma operator performs right to left, the pair on right would be overwritten with the pair with same key on the left.

```
std::multimap < int, std::string > mmp { std::make_pair(2, "stackoverflow"),
                                     std::make_pair(1, "docs-beta"),
                                     std::make_pair(2, "stackexchange")  };
// 1 docs-beta
// 2 stackoverflow
// 2 stackexchange

std::map < int, std::string > mp {  std::make_pair(2, "stackoverflow"),
                                std::make_pair(1, "docs-beta"),
                                std::make_pair(2, "stackexchange")  };
// 1 docs-beta
// 2 stackoverflow
```

Both could be initialized with iterator.

```
// From std::map or std::multimap iterator
std::multimap< int , int > mmp{ {1, 2}, {3, 4}, {6, 5}, {8, 9}, {6, 8}, {3, 4},
                               {6, 7} };
                       // {1, 2}, {3, 4}, {3, 4}, {6, 5}, {6, 8}, {6, 7}, {8, 9}
auto it = mmp.begin();
std::advance(it,3); //moved cursor on first {6, 5}
std::map< int, int > mp(it, mmp.end()); // {6, 5}, {8, 9}

//From std::pair array
std::pair< int, int > arr[10];
arr[0] = {1, 3};
arr[1] = {1, 5};
arr[2] = {2, 5};
arr[3] = {0, 1};
std::map< int, int > mp(arr,arr+4); //{0 , 1}, {1, 3}, {2, 5}

//From std::vector of std::pair
std::vector< std::pair<int, int> > v{ {1, 5}, {5, 1}, {3, 6}, {3, 2} };
std::multimap< int, int > mp(v.begin(), v.end());
                        // {1, 5}, {3, 6}, {3, 2}, {5, 1}
```

Section 50.5: Checking number of elements

The container `std::map` has a member function `empty()`, which returns `true` or `false`, depending on whether the map is empty or not. The member function `size()` returns the number of element stored in a `std::map` container:

```
std::map<std::string , int> rank {{"facebook.com", 1} ,{"google.com", 2}, {"youtube.com", 3}};
if(!rank.empty()){
    std::cout << "Number of elements in the rank map: " << rank.size() << std::endl;
}
else{
    std::cout << "The rank map is empty" << std::endl;
}
```

Section 50.6: Types of Maps

Regular Map

A map is an associative container, containing key-value pairs.

```
#include <string>
#include <map>
std::map<std::string, size_t> fruits_count;
```

In the above example, `std::string` is the *key* type, and `size_t` is a *value*.

The key acts as an index in the map. Each key must be unique, and must be ordered.

- If you need mutliple elements with the same key, consider using `multimap` (explained below)
- If your value type does not specify any ordering, or you want to override the default ordering, you may provide one:

```
#include <string>
#include <map>
```

```
#include <cstring>
struct StrLess {
    bool operator()(const std::string& a, const std::string& b) {
        return strncmp(a.c_str(), b.c_str(), 8)<0;
               //compare only up to 8 first characters
    }
}
std::map<std::string, size_t, StrLess> fruits_count2;
```

If `StrLess` comparator returns `false` for two keys, they are considered the same even if their actual contents differ.

Multi-Map

Multimap allows multiple key-value pairs with the same key to be stored in the map. Otherwise, its interface and creation is very similar to the regular map.

```
#include <string>
#include <map>
std::multimap<std::string, size_t> fruits_count;
std::multimap<std::string, size_t, StrLess> fruits_count2;
```

Hash-Map (Unordered Map)

A hash map stores key-value pairs similar to a regular map. It does not order the elements with respect to the key though. Instead, a hash value for the key is used to quickly access the needed key-value pairs.

```
#include <string>
#include <unordered_map>
std::unordered_map<std::string, size_t> fruits_count;
```

Unordered maps are usually faster, but the elements are not stored in any predictable order. For example, iterating over all elements in an `unordered_map` gives the elements in a seemingly random order.

Section 50.7: Deleting elements

Removing all elements:

```
std::multimap< int , int > mmp{ {1, 2}, {3, 4}, {6, 5}, {8, 9}, {3, 4}, {6, 7} };
mmp.clear(); //empty multimap
```

Removing element from somewhere with the help of iterator:

```
std::multimap< int , int > mmp{ {1, 2}, {3, 4}, {6, 5}, {8, 9}, {3, 4}, {6, 7} };
                            // {1, 2}, {3, 4}, {3, 4}, {6, 5}, {6, 7}, {8, 9}
auto it = mmp.begin();
std::advance(it,3); // moved cursor on first {6, 5}
mmp.erase(it); // {1, 2}, {3, 4}, {3, 4}, {6, 7}, {8, 9}
```

Removing all elements in a range:

```
std::multimap< int , int > mmp{ {1, 2}, {3, 4}, {6, 5}, {8, 9}, {3, 4}, {6, 7} };
                            // {1, 2}, {3, 4}, {3, 4}, {6, 5}, {6, 7}, {8, 9}
auto it = mmp.begin();
auto it2 = it;
```

```
it++; //moved first cursor on first {3, 4}
std::advance(it2,3);  //moved second cursor on first {6, 5}
mmp.erase(it,it2); // {1, 2}, {6, 5}, {6, 7}, {8, 9}
```

Removing all elements having a provided value as key:

```
std::multimap< int , int > mmp{ {1, 2}, {3, 4}, {6, 5}, {8, 9}, {3, 4}, {6, 7} };
                            // {1, 2}, {3, 4}, {3, 4}, {6, 5}, {6, 7}, {8, 9}
mmp.erase(6); // {1, 2}, {3, 4}, {3, 4}, {8, 9}
```

Removing elements that satisfy a predicate `pred`:

```
std::map<int,int> m;
auto it = m.begin();
while (it != m.end())
{
   if (pred(*it))
       it = m.erase(it);
   else
       ++it;
}
```

Section 50.8: Iterating over std::map or std::multimap

`std::map` or `std::multimap` could be traversed by the following ways:

```
std::multimap< int , int > mmp{ {1, 2}, {3, 4}, {6, 5}, {8, 9}, {3, 4}, {6, 7} };

//Range based loop - since C++11
for(const auto &x: mmp)
    std::cout<< x.first <<":"<< x.second << std::endl;

//Forward iterator for loop: it would loop through first element to last element
//it will be a std::map< int, int >::iterator
for (auto it = mmp.begin(); it != mmp.end(); ++it)
std::cout<< it->first <<":"<< it->second << std::endl; //Do something with iterator

//Backward iterator for loop: it would loop through last element to first element
//it will be a std::map< int, int >::reverse_iterator
for (auto it = mmp.rbegin(); it != mmp.rend(); ++it)
std::cout<< it->first <<" "<< it->second << std::endl; //Do something with iterator
```

While iterating over a `std::map` or a `std::multimap`, the use of `auto` is preferred to avoid useless implicit conversions (see this SO answer for more details).

Section 50.9: Creating std::map with user-defined types as key

In order to be able to use a class as the key in a map, all that is required of the key is that it be `copiable` and `assignable`. The ordering within the map is defined by the third argument to the template (and the argument to the constructor, if used). This *defaults* to `std::less<KeyType>`, which defaults to the `<` operator, but there's no requirement to use the defaults. Just write a comparison operator (preferably as a functional object):

```
struct CmpMyType
{
    bool operator()( MyType const& lhs, MyType const& rhs ) const
    {
```

```
        // ...
    }
};
```

In C++, the "compare" predicate must be a strict weak ordering. In particular, `compare(X,X)` must return `false` for any `X`. i.e. if `CmpMyType()(a, b)` returns true, then `CmpMyType()(b, a)` must return false, and if both return false, the elements are considered equal (members of the same equivalence class).

Strict Weak Ordering

This is a mathematical term to define a relationship between two objects.
Its definition is:

> Two objects x and y are equivalent if both f(x, y) and f(y, x) are false. Note that an object is always (by the irreflexivity invariant) equivalent to itself.

In terms of C++ this means if you have two objects of a given type, you should return the following values when compared with the operator <.

```
X    a;
X    b;

Condition:                  Test:     Result
a is equivalent to b:       a < b     false
a is equivalent to b        b < a     false

a is less than b            a < b     true
a is less than b            b < a     false

b is less than a            a < b     false
b is less than a            b < a     true
```

How you define equivalent/less is totally dependent on the type of your object.

Chapter 51: std::optional

Section 51.1: Using optionals to represent the absence of a value

Before C++17, having pointers with a value of `nullptr` commonly represented the absence of a value. This is a good solution for large objects that have been dynamically allocated and are already managed by pointers. However, this solution does not work well for small or primitive types such as `int`, which are rarely ever dynamically allocated or managed by pointers. `std::optional` provides a viable solution to this common problem.

In this example, `struct Person` is defined. It is possible for a person to have a pet, but not necessary. Therefore, the `pet` member of `Person` is declared with an `std::optional` wrapper.

```
#include <iostream>
#include <optional>
#include <string>

struct Animal {
    std::string name;
};

struct Person {
    std::string name;
    std::optional<Animal> pet;
};

int main() {
    Person person;
    person.name = "John";

    if (person.pet) {
        std::cout << person.name << "'s pet's name is " <<
            person.pet->name << std::endl;
    }
    else {
        std::cout << person.name << " is alone." << std::endl;
    }
}
```

Section 51.2: optional as return value

```
std::optional<float> divide(float a, float b) {
  if (b!=0.f) return a/b;
  return {};
}
```

Here we return either the fraction `a/b`, but if it is not defined (would be infinity) we instead return the empty optional.

A more complex case:

```
template<class Range, class Pred>
auto find_if( Range&& r, Pred&& p ) {
  using std::begin; using std::end;
  auto b = begin(r), e = end(r);
  auto r = std::find_if(b, e , p );
  using iterator = decltype(r);
```

```
  if (r==e)
    return std::optional<iterator>();
  return std::optional<iterator>(r);
}
template<class Range, class T>
auto find( Range&& r, T const& t ) {
  return find_if( std::forward<Range>(r), [&t](auto&& x){return x==t;} );
}
```

`find( some_range, 7 )` searches the container or range `some_range` for something equal to the number 7. `find_if` does it with a predicate.

It returns either an empty optional if it was not found, or an optional containing an iterator tothe element if it was.

This allows you to do:

```
if (find( vec, 7 )) {
  // code
}
```

or even

```
if (auto oit = find( vec, 7 )) {
  vec.erase(*oit);
}
```

without having to mess around with begin/end iterators and tests.

Section 51.3: value_or

```
void print_name( std::ostream& os, std::optional<std::string> const& name ) {
  std::cout "Name is: " << name.value_or("<name missing>") << '\n';
}
```

`value_or` either returns the value stored in the optional, or the argument if there is nothing store there.

This lets you take the maybe-null optional and give a default behavior when you actually need a value. By doing it this way, the "default behavior" decision can be pushed back to the point where it is best made and immediately needed, instead of generating some default value deep in the guts of some engine.

Section 51.4: Introduction

Optionals (also known as Maybe types) are used to represent a type whose contents may or may not be present. They are implemented in C++17 as the `std::optional` class. For example, an object of type `std::optional<int>` may contain some value of type `int`, or it may contain no value.

Optionals are commonly used either to represent a value that may not exist or as a return type from a function that can fail to return a meaningful result.

Other approaches to optional

There are many other approach to solving the problem that `std::optional` solves, but none of them are quite complete: using a pointer, using a sentinel, or using a `pair<bool, T>`.

Optional vs Pointer

In some cases, we can provide a pointer to an existing object or `nullptr` to indicate failure. But this is limited to those cases where objects already exist - `optional`, as a value type, can also be used to return new objects without resorting to memory allocation.

Optional vs Sentinel

A common idiom is to use a special value to indicate that the value is meaningless. This may be 0 or -1 for integral types, or `nullptr` for pointers. However, this reduces the space of valid values (you cannot differentiate between a valid 0 and a meaningless 0) and many types do not have a natural choice for the sentinel value.

Optional vs `std::pair<bool, T>`

Another common idiom is to provide a pair, where one of the elements is a `bool` indicating whether or not the value is meaningful.

This relies upon the value type being default-constructible in the case of error, which is not possible for some types and possible but undesirable for others. An `optional<T>`, in the case of error, does not need to construct anything.

Section 51.5: Using optionals to represent the failure of a function

Before C++17, a function typically represented failure in one of several ways:

- A null pointer was returned.
  - e.g. Calling a function `Delegate *App::get_delegate()` on an `App` instance that did not have a delegate would return `nullptr`.
  - This is a good solution for objects that have been dynamically allocated or are large and managed by pointers, but isn't a good solution for small objects that are typically stack-allocated and passed by copying.
- A specific value of the return type was reserved to indicate failure.
  - e.g. Calling a function `unsigned shortest_path_distance(Vertex a, Vertex b)` on two vertices that are not connected may return zero to indicate this fact.
- The value was paired together with a `bool` to indicate is the returned value was meaningful.
  - e.g. Calling a function `std::pair<int, bool> parse(const std::string &str)` with a string argument that is not an integer would return a pair with an undefined `int` and a `bool` set to `false`.

In this example, John is given two pets, Fluffy and Furball. The function `Person::pet_with_name()` is then called to retrieve John's pet Whiskers. Since John does not have a pet named Whiskers, the function fails and `std::nullopt` is returned instead.

```
#include <iostream>
#include <optional>
#include <string>
#include <vector>

struct Animal {
    std::string name;
};

struct Person {
    std::string name;
    std::vector<Animal> pets;

    std::optional<Animal> pet_with_name(const std::string &name) {
        for (const Animal &pet : pets) {
```

```
            if (pet.name == name) {
                return pet;
            }
        }
        return std::nullopt;
    }
};

int main() {
    Person john;
    john.name = "John";

    Animal fluffy;
    fluffy.name = "Fluffy";
    john.pets.push_back(fluffy);

    Animal furball;
    furball.name = "Furball";
    john.pets.push_back(furball);

    std::optional<Animal> whiskers = john.pet_with_name("Whiskers");
    if (whiskers) {
        std::cout << "John has a pet named Whiskers." << std::endl;
    }
    else {
        std::cout << "Whiskers must not belong to John." << std::endl;
    }
}
```

Chapter 52: std::function: To wrap any element that is callable

Section 52.1: Simple usage

```
#include <iostream>
#include <functional>
std::function<void(int , const std::string&)> myFuncObj;
void theFunc(int i, const std::string& s)
{
    std::cout << s << ": " << i << std::endl;
}
int main(int argc, char *argv[])
{
    myFuncObj = theFunc;
    myFuncObj(10, "hello world");
}
```

Section 52.2: std::function used with std::bind

Think about a situation where we need to callback a function with arguments. `std::function` used with `std::bind` gives a very powerful design construct as shown below.

```
class A
{
public:
    std::function<void(int, const std::string&)> m_CbFunc = nullptr;
    void foo()
    {
        if (m_CbFunc)
        {
            m_CbFunc(100, "event fired");
        }
    }

};

class B
{
public:
    B()
    {
        auto aFunc = std::bind(&B::eventHandler, this, std::placeholders::_1,
std::placeholders::_2);
        anObjA.m_CbFunc = aFunc;
    }
    void eventHandler(int i, const std::string& s)
    {
        std::cout << s << ": " << i << std::endl;
    }

    void DoSomethingOnA()
    {
        anObjA.foo();
    }

    A anObjA;
};
```

```
int main(int argc, char *argv[])
{
     B anObjB;
     anObjB.DoSomethingOnA();
}
```

Section 52.3: Binding std::function to a different callable types

```
/*
 * This example show some ways of using std::function to call
 *  a) C-like function
 *  b) class-member function
 *  c) operator()
 *  d) lambda function
 *
 * Function call can be made:
 *  a) with right arguments
 *  b) argumens with different order, types and count
 */
#include <iostream>
#include <functional>
#include <iostream>
#include <vector>

using std::cout;
using std::endl;
using namespace std::placeholders;



// simple function to be called
double foo_fn(int x, float y, double z)
{
  double res = x + y + z;
  std::cout << "foo_fn called with arguments: "
            << x << ", " << y << ", " << z
            << " result is : " << res
            << std::endl;
  return res;
}

// structure with member function to call
struct foo_struct
{
    // member function to call
    double foo_fn(int x, float y, double z)
    {
        double res = x + y + z;
        std::cout << "foo_struct::foo_fn called with arguments: "
                << x << ", " << y << ", " << z
                << " result is : " << res
                << std::endl;
        return res;
    }
    // this member function has different signature - but it can be used too
    // please not that argument order is changed too
    double foo_fn_4(int x, double z, float y, long xx)
    {
        double res = x + y + z + xx;
        std::cout << "foo_struct::foo_fn_4 called with arguments: "
```

```
                << x << ", " << z << ", " << y << ", " << xx
                << " result is : " << res
                << std::endl;
        return res;
    }
    // overloaded operator() makes whole object to be callable
    double operator()(int x, float y, double z)
    {
        double res = x + y + z;
        std::cout << "foo_struct::operator() called with arguments: "
                << x << ", " << y << ", " << z
                << " result is : " << res
                << std::endl;
        return res;
    }
};


int main(void)
{
  // typedefs
  using function_type = std::function<double(int, float, double)>;

  // foo_struct instance
  foo_struct fs;

  // here we will store all binded functions
  std::vector<function_type> bindings;

  // var #1 - you can use simple function
  function_type var1 = foo_fn;
  bindings.push_back(var1);

  // var #2 - you can use member function
  function_type var2 = std::bind(&foo_struct::foo_fn, fs, _1, _2, _3);
  bindings.push_back(var2);

  // var #3 - you can use member function with different signature
  // foo_fn_4 has different count of arguments and types
  function_type var3 = std::bind(&foo_struct::foo_fn_4, fs, _1, _3, _2, 0l);
  bindings.push_back(var3);

  // var #4 - you can use object with overloaded operator()
  function_type var4 = fs;
  bindings.push_back(var4);

  // var #5 - you can use lambda function
  function_type var5 = [](int x, float y, double z)
    {
        double res = x + y + z;
        std::cout << "lambda  called with arguments: "
                << x << ", " << y << ", " << z
                << " result is : " << res
                << std::endl;
        return res;
    };
  bindings.push_back(var5);

  std::cout << "Test stored functions with arguments: x = 1, y = 2, z = 3"
            << std::endl;

  for (auto f : bindings)
```

```
    f(1, 2, 3);

}
```

Live

Output:

```
Test stored functions with arguments: x = 1, y = 2, z = 3
foo_fn called with arguments: 1, 2, 3 result is : 6
foo_struct::foo_fn called with arguments: 1, 2, 3 result is : 6
foo_struct::foo_fn_4 called with arguments: 1, 3, 2, 0 result is : 6
foo_struct::operator() called with arguments: 1, 2, 3 result is : 6
lambda  called with arguments: 1, 2, 3 result is : 6
```

Section 52.4: Storing function arguments in std::tuple

Some programs need so store arguments for future calling of some function.

This example shows how to call any function with arguments stored in std::tuple

```
#include <iostream>
#include <functional>
#include <tuple>
#include <iostream>

// simple function to be called
double foo_fn(int x, float y, double z)
{
   double res =  x + y + z;
   std::cout << "foo_fn called. x = " << x << " y = " << y << " z = " << z
             << " res=" << res;
   return res;
}

// helpers for tuple unrolling
template<int ...> struct seq {};
template<int N, int ...S> struct gens : gens<N-1, N-1, S...> {};
template<int ...S> struct gens<0, S...>{ typedef seq<S...> type; };

// invocation helper
template<typename FN, typename P, int ...S>
double call_fn_internal(const FN& fn, const P& params, const seq<S...>)
{
   return fn(std::get<S>(params) ...);
}
// call function with arguments stored in std::tuple
template<typename Ret, typename ...Args>
Ret call_fn(const std::function<Ret(Args...)>& fn,
            const std::tuple<Args...>& params)
{
    return call_fn_internal(fn, params, typename gens<sizeof...(Args)>::type());
}


int main(void)
{
  // arguments
  std::tuple<int, float, double> t = std::make_tuple(1, 5, 10);
  // function to call
```

```
    std::function<double(int, float, double)> fn = foo_fn;

    // invoke a function with stored arguments
    call_fn(fn, t);
}
```

Live

Output:

```
foo_fn called. x = 1 y = 5 z = 10 res=16
```

Section 52.5: std::function with lambda and std::bind

```
#include <iostream>
#include <functional>

using std::placeholders::_1; // to be used in std::bind example

int stdf_foobar (int x, std::function<int(int)> moo)
{
    return x + moo(x); // std::function moo called
}

int foo (int x) { return 2+x; }

int foo_2 (int x, int y) { return 9*x + y; }

int main()
{
    int a = 2;

    /* Function pointers */
    std::cout << stdf_foobar(a, &foo) << std::endl; // 6 ( 2 + (2+2) )
    // can also be: stdf_foobar(2, foo)

    /* Lambda expressions */
    /* An unnamed closure from a lambda expression can be
     * stored in a std::function object:
     */
    int capture_value = 3;
    std::cout << stdf_foobar(a,
                             [capture_value](int param) -> int { return 7 + capture_value * param;
})
              << std::endl;
    // result: 15 ==  value + (7 * capture_value * value) == 2 + (7 + 3 * 2)

    /* std::bind expressions */
    /* The result of a std::bind expression can be passed.
     * For example by binding parameters to a function pointer call:
     */
    int b = stdf_foobar(a, std::bind(foo_2, _1, 3));
    std::cout << b << std::endl;
    // b == 23 == 2 + ( 9*2 + 3 )
    int c = stdf_foobar(a, std::bind(foo_2, 5, _1));
    std::cout << c << std::endl;
    // c == 49 == 2 + ( 9*5 + 2 )

    return 0;
}
```

Section 52.6: `function` overhead

std::function can cause significant overhead. Because std::function has [value semantics][1], it must copy or move the given callable into itself. But since it can take callables of an arbitrary type, it will frequently have to allocate memory dynamically to do this.

Some function implementations have so-called "small object optimization", where small types (like function pointers, member pointers, or functors with very little state) will be stored directly in the function object. But even this only works if the type is noexcept move constructible. Furthermore, the C++ standard does not require that all implementations provide one.

Consider the following:

```
//Header file
using MyPredicate = std::function<bool(const MyValue &, const MyValue &)>;

void SortMyContainer(MyContainer &C, const MyPredicate &pred);

//Source file
void SortMyContainer(MyContainer &C, const MyPredicate &pred)
{
    std::sort(C.begin(), C.end(), pred);
}
```

A template parameter would be the preferred solution for SortMyContainer, but let us assume that this is not possible or desirable for whatever reason. SortMyContainer does not need to store pred beyond its own call. And yet, pred may well allocate memory if the functor given to it is of some non-trivial size.

function allocates memory because it needs something to copy/move into; function takes ownership of the callable it is given. But SortMyContainer does not need to *own* the callable; it's just referencing it. So using function here is overkill; it may be efficient, but it may not.

There is no standard library function type that merely references a callable. So an alternate solution will have to be found, or you can choose to live with the overhead.

Also, function has no effective means to control where the memory allocations for the object come from. Yes, it has constructors that take an allocator, but [many implementations do not implement them correctly... or even *at all*][2].

Version ≥ C++17

The function constructors that take an allocator no longer are part of the type. Therefore, there is no way to manage the allocation.

Calling a function is also slower than calling the contents directly. Since any function instance could hold any callable, the call through a function must be indirect. The overhead of calling function is on the order of a virtual function call.

Chapter 53: std::forward_list

`std::forward_list` is a container that supports fast insertion and removal of elements from anywhere in the container. Fast random access is not supported. It is implemented as a singly-linked list and essentially does not have any overhead compared to its implementation in C. Compared to `std::list` this container provides more space efficient storage when bidirectional iteration is not needed.

Section 53.1: Example

```
#include <forward_list>
#include <string>
#include <iostream>

template<typename T>
std::ostream& operator<<(std::ostream& s, const std::forward_list<T>& v) {
    s.put('[');
    char comma[3] = {'\0', ' ', '\0'};
    for (const auto& e : v) {
        s << comma << e;
        comma[0] = ',';
    }
    return s << ']';
}

int main()
{
    // c++11 initializer list syntax:
    std::forward_list<std::string> words1 {"the", "frogurt", "is", "also", "cursed"};
    std::cout << "words1: " << words1 << '\n';

    // words2 == words1
    std::forward_list<std::string> words2(words1.begin(), words1.end());
    std::cout << "words2: " << words2 << '\n';

    // words3 == words1
    std::forward_list<std::string> words3(words1);
    std::cout << "words3: " << words3 << '\n';

    // words4 is {"Mo", "Mo", "Mo", "Mo", "Mo"}
    std::forward_list<std::string> words4(5, "Mo");
    std::cout << "words4: " << words4 << '\n';
}
```

Output:

```
words1: [the, frogurt, is, also, cursed]
words2: [the, frogurt, is, also, cursed]
words3: [the, frogurt, is, also, cursed]
words4: [Mo, Mo, Mo, Mo, Mo]
```

Section 53.2: Methods

Method name	Definition
`operator=`	assigns values to the container
`assign`	assigns values to the container
`get_allocator`	returns the associated allocator

------	------
Element access	
`front`	access the first element
------	------
Iterators	
`before_begin`	returns an iterator to the element before beginning
`cbefore_begin`	returns a constant iterator to the element before beginning
`begin`	returns an iterator to the beginning
`cbegin`	returns a const iterator to the beginning
`end`	returns an iterator to the end
`cend`	returns a iterator to the end
Capacity	
`empty`	checks whether the container is empty
`max_size`	returns the maximum possible number of elements
Modifiers	
`clear`	clears the contents
`insert_after`	inserts elements after an element
`emplace_after`	constructs elements in-place after an element
`erase_after`	erases an element after an element
`push_front`	inserts an element to the beginning
`emplace_front`	constructs an element in-place at the beginning
`pop_front`	removes the first element
`resize`	changes the number of elements stored
`swap`	swaps the contents
Operations	
`merge`	merges two sorted lists
`splice_after`	moves elements from another forward_list
`remove`	removes elements satisfying specific criteria
`remove_if`	removes elements satisfying specific criteria
`reverse`	reverses the order of the elements
`unique`	removes consecutive duplicate elements
`sort`	sorts the elements

Chapter 54: std::pair

Section 54.1: Compare operators

Parameters of these operators are `lhs` and `rhs`

- `operator==` tests if both elements on `lhs` and `rhs` pair are equal. The return value is `true` if both `lhs.first == rhs.first` AND `lhs.second == rhs.second`, otherwise `false`

```
std::pair<int, int> p1 = std::make_pair(1, 2);
std::pair<int, int> p2 = std::make_pair(2, 2);

if (p1 == p2)
    std::cout << "equals";
else
    std::cout << "not equal"//statement will show this, because they are not identical
```

- `operator!=` tests if any elements on `lhs` and `rhs` pair are not equal. The return value is `true` if either `lhs.first != rhs.first` OR `lhs.second != rhs.second`, otherwise return `false`.
- `operator<` tests if `lhs.first<rhs.first`, returns `true`. Otherwise, if `rhs.first<lhs.first` returns `false`. Otherwise, if `lhs.second<rhs.second` returns `true`, otherwise, returns `false`.
- `operator<=` returns `!(rhs<lhs)`
- `operator>` returns `rhs<lhs`
- `operator>=` returns `!(lhs<rhs)`

  Another example with containers of pairs. It uses `operator<` because it needs to sort container.

```
#include <iostream>
#include <utility>
#include <vector>
#include <algorithm>
#include <string>

int main()
{
    std::vector<std::pair<int, std::string>> v = { {2, "baz"},
                                                   {2, "bar"},
                                                   {1, "foo"} };
    std::sort(v.begin(), v.end());

    for(const auto& p: v) {
        std::cout << "(" << p.first << "," << p.second << ") ";
        //output: (1,foo) (2,bar) (2,baz)
    }
}
```

Section 54.2: Creating a Pair and accessing the elements

Pair allows us to treat two objects as one object. Pairs can be easily constructed with the help of template function `std::make_pair`.

Alternative way is to create pair and assign its elements (`first` and `second`) later.

```
#include <iostream>
#include <utility>

int main()
{
    std::pair<int,int> p = std::make_pair(1,2); //Creating the pair
    std::cout << p.first << " " << p.second << std::endl; //Accessing the elements



    //We can also create a pair and assign the elements later
    std::pair<int,int> p1;
    p1.first = 3;
    p1.second = 4;
    std::cout << p1.first << " " << p1.second << std::endl;

    //We can also create a pair using a constructor
    std::pair<int,int> p2 = std::pair<int,int>(5, 6);
    std::cout << p2.first << " " << p2.second << std::endl;

    return 0;
}
```

Chapter 55: std::atomics

Section 55.1: atomic types

Each instantiation and full specialization of the `std::atomic` template defines an atomic type. If one thread writes to an atomic object while another thread reads from it, the behavior is well-defined (see memory model for details on data races)

In addition, accesses to atomic objects may establish inter-thread synchronization and order non-atomic memory accesses as specified by `std::memory_order`.

std::atomic may be instantiated with any `TriviallyCopyable type T.` `std::atomic` is neither copyable nor movable.

The standard library provides specializations of the std::atomic template for the following types:

1. One full specialization for the type `bool` and its typedef name is defined that is treated as a non-specialized `std::atomic<T>` except that it has standard layout, trivial default constructor, trivial destructors, and supports aggregate initialization syntax:

Typedef name	Full specialization
`std::atomic_bool`	`std::atomic<bool>`

2)Full specializations and typedefs for integral types, as follows:

Typedef name	Full specialization
`std::atomic_char`	`std::atomic<char>`
`std::atomic_char`	`std::atomic<char>`
`std::atomic_schar`	`std::atomic<signed char>`
`std::atomic_uchar`	`std::atomic<unsigned char>`
`std::atomic_short`	`std::atomic<short>`
`std::atomic_ushort`	`std::atomic<unsigned short>`
`std::atomic_int`	`std::atomic<int>`
`std::atomic_uint`	`std::atomic<unsigned int>`
`std::atomic_long`	`std::atomic<long>`
`std::atomic_ulong`	`std::atomic<unsigned long>`
`std::atomic_llong`	`std::atomic<long long>`
`std::atomic_ullong`	`std::atomic<unsigned long long>`
`std::atomic_char16_t`	`std::atomic<char16_t>`
`std::atomic_char32_t`	`std::atomic<char32_t>`
`std::atomic_wchar_t`	`std::atomic<wchar_t>`
`std::atomic_int8_t`	`std::atomic<std::int8_t>`
`std::atomic_uint8_t`	`std::atomic<std::uint8_t>`
`std::atomic_int16_t`	`std::atomic<std::int16_t>`
`std::atomic_uint16_t`	`std::atomic<std::uint16_t>`
`std::atomic_int32_t`	`std::atomic<std::int32_t>`
`std::atomic_uint32_t`	`std::atomic<std::uint32_t>`
`std::atomic_int64_t`	`std::atomic<std::int64_t>`
`std::atomic_uint64_t`	`std::atomic<std::uint64_t>`
`std::atomic_int_least8_t`	`std::atomic<std::int_least8_t>`
`std::atomic_uint_least8_t`	`std::atomic<std::uint_least8_t>`

std::atomic_int_least16_t	std::atomic<std::int_least16_t>
std::atomic_uint_least16_t	std::atomic<std::uint_least16_t>
std::atomic_int_least32_t	std::atomic<std::int_least32_t>
std::atomic_uint_least32_t	std::atomic<std::uint_least32_t>
std::atomic_int_least64_t	std::atomic<std::int_least64_t>
std::atomic_uint_least64_t	std::atomic<std::uint_least64_t>
std::atomic_int_fast8_t	std::atomic<std::int_fast8_t>
std::atomic_uint_fast8_t	std::atomic<std::uint_fast8_t>
std::atomic_int_fast16_t	std::atomic<std::int_fast16_t>
std::atomic_uint_fast16_t	std::atomic<std::uint_fast16_t>
std::atomic_int_fast32_t	std::atomic<std::int_fast32_t>
std::atomic_uint_fast32_t	std::atomic<std::uint_fast32_t>
std::atomic_int_fast64_t	std::atomic<std::int_fast64_t>
std::atomic_uint_fast64_t	std::atomic<std::uint_fast64_t>
std::atomic_intptr_t	std::atomic<std::intptr_t>
std::atomic_uintptr_t	std::atomic<std::uintptr_t>
std::atomic_size_t	std::atomic<std::size_t>
std::atomic_ptrdiff_t	std::atomic<std::ptrdiff_t>
std::atomic_intmax_t	std::atomic<std::intmax_t>
std::atomic_uintmax_t	std::atomic<std::uintmax_t>

Simple example of using std::atomic_int

```
#include <iostream>       // std::cout
#include <atomic>         // std::atomic, std::memory_order_relaxed
#include <thread>         // std::thread

std::atomic_int foo (0);

void set_foo(int x) {
  foo.store(x,std::memory_order_relaxed);     // set value atomically
}

void print_foo() {
  int x;
  do {
    x = foo.load(std::memory_order_relaxed);  // get value atomically
  } while (x==0);
  std::cout << "foo: " << x << '\n';
}

int main ()
{
  std::thread first (print_foo);
  std::thread second (set_foo,10);
  first.join();
  //second.join();
  return 0;
}
//output: foo: 10
```

Chapter 56: std::variant

Section 56.1: Create pseudo-method pointers

This is an advanced example.

You can use variant for light weight type erasure.

```
template<class F>
struct pseudo_method {
  F f;
  // enable C++17 class type deduction:
  pseudo_method( F&& fin ):f(std::move(fin)) {}

  // Koenig lookup operator->*, as this is a pseudo-method it is appropriate:
  template<class Variant> // maybe add SFINAE test that LHS is actually a variant.
  friend decltype(auto) operator->*( Variant&& var, pseudo_method const& method ) {
    // var->*method returns a lambda that perfect forwards a function call,
    // behaving like a method pointer basically:
    return [&](auto&&...args)->decltype(auto) {
      // use visit to get the type of the variant:
      return std::visit(
        [&](auto&& self)->decltype(auto) {
          // decltype(x)(x) is perfect forwarding in a lambda:
          return method.f( decltype(self)(self), decltype(args)(args)... );
        },
        std::forward<Var>(var)
      );
    };
  }
};
```

this creates a type that overloads `operator->*` with a `Variant` on the left hand side.

```
// C++17 class type deduction to find template argument of `print` here.
// a pseudo-method lambda should take `self` as its first argument, then
// the rest of the arguments afterwards, and invoke the action:
pseudo_method print = [](auto&& self, auto&&...args)->decltype(auto) {
  return decltype(self)(self).print( decltype(args)(args)... );
};
```

Now if we have 2 types each with a `print` method:

```
struct A {
  void print( std::ostream& os ) const {
    os << "A";
  }
};
struct B {
  void print( std::ostream& os ) const {
    os << "B";
  }
};
```

note that they are unrelated types. We can:

```
std::variant<A,B> var = A{};
```

```
(var->*print)(std::cout);
```

and it will dispatch the call directly to `A::print(std::cout)` for us. If we instead initialized the `var` with `B{}`, it would dispatch to `B::print(std::cout)`.

If we created a new type C:

```
struct C {};
```

then:

```
std::variant<A,B,C> var = A{};
(var->*print)(std::cout);
```

will fail to compile, because there is no `C.print(std::cout)` method.

Extending the above would permit free function `print`s to be detected and used, possibly with use of `if constexpr` within the `print` pseudo-method.

live example currently using `boost::variant` in place of `std::variant`.

Section 56.2: Basic std::variant use

This creates a variant (a tagged union) that can store either an `int` or a `string`.

```
std::variant< int, std::string > var;
```

We can store one of either type in it:

```
var = "hello"s;
```

And we can access the contents via `std::visit`:

```
// Prints "hello\n":
visit( [](auto&& e) {
  std::cout << e << '\n';
}, var );
```

by passing in a polymorphic lambda or similar function object.

If we are certain we know what type it is, we can get it:

```
auto str = std::get<std::string>(var);
```

but this will throw if we get it wrong. `get_if`:

```
auto* str  = std::get_if<std::string>(&var);
```

returns `nullptr` if you guess wrong.

Variants guarantee no dynamic memory allocation (other than which is allocated by their contained types). Only one of the types in a variant is stored there, and in rare cases (involving exceptions while assigning and no safe way to back out) the variant can become empty.

Variants let you store multiple value types in one variable safely and efficiently. They are basically smart, type-safe

unions.

Section 56.3: Constructing a `std::variant`

This does not cover allocators.

```
struct A {};
struct B { B()=default; B(B const&)=default; B(int){}; };
struct C { C()=delete; C(int) {}; C(C const&)=default; };
struct D { D( std::initializer_list<int> ) {}; D(D const&)=default; D()=default; };

std::variant<A,B> var_ab0; // contains a A()
std::variant<A,B> var_ab1 = 7; // contains a B(7)
std::variant<A,B> var_ab2 = var_ab1; // contains a B(7)
std::variant<A,B,C> var_abc0{ std::in_place_type<C>, 7 }; // contains a C(7)
std::variant<C> var_c0; // illegal, no default ctor for C
std::variant<A,D> var_ad0( std::in_place_type<D>, {1,3,3,4} ); // contains D{1,3,3,4}
std::variant<A,D> var_ad1( std::in_place_index<0> ); // contains A{}
std::variant<A,D> var_ad2( std::in_place_index<1>, {1,3,3,4} ); // contains D{1,3,3,4}
```

Chapter 57: std::iomanip

Section 57.1: std::setprecision

When used in an expression `out << setprecision(n)` or `in >> setprecision(n)`, sets the precision parameter of the stream out or in to exactly n. Parameter of this function is integer, which is new value for precision.

Example:

```
#include <iostream>
#include <iomanip>
#include <cmath>
#include <limits>
int main()
{
    const long double pi = std::acos(-1.L);
    std::cout << "default precision (6): " << pi << '\n'
              << "std::precision(10):    " << std::setprecision(10) << pi << '\n'
              << "max precision:         "
              << std::setprecision(std::numeric_limits<long double>::digits10 + 1)
              << pi << '\n';
}
//Output
//default precision (6): 3.14159
//std::precision(10):    3.141592654
//max precision:         3.141592653589793239
```

Section 57.2: std::setfill

When used in an expression `out << setfill(c)` sets the fill character of the stream out to `c`.

Note: The current fill character may be obtained with `std::ostream::fill`.

Example:

```
#include <iostream>
#include <iomanip>
int main()
{
    std::cout << "default fill: " << std::setw(10) << 42 << '\n'
          << "setfill('*'): " << std::setfill('*')
                              << std::setw(10) << 42 << '\n';
}
//output::
//default fill:         42
//setfill('*'): ********42
```

Section 57.3: std::setiosflags

When used in an expression `out << setiosflags(mask)` or `in >> setiosflags(mask)`, sets all format flags of the stream out or in as specified by the mask.

List of all `std::ios_base::fmtflags` :

- `dec` - use decimal base for integer I/O
- `oct` - use octal base for integer I/O

- `hex` - use hexadecimal base for integer I/O
- `basefield` - `dec|oct|hex|0` useful for masking operations
- `left` - left adjustment(add fill characters to the right)
- `right` - right adjustment (adds fill characters to the left)
- `internal` - internal adjustment (adds fill characters to the internal designated point)
- `adjustfield` - `left|right|internal`. Useful for masking operations
- `scientific` - generate floating point types using scientific notation, or hex notation if combined with fixed
- `fixed` - generate floating point types using fixed notation, or hex notation if combined with scientific
- `floatfield` - `scientific|fixed|(scientific|fixed)|0`. Useful for masking operations
- `boolalpha` - insert and extract `bool` type in alphanumeric format
- `showbase` - generate a prefix indicating the numeric base for integer output, require the currency indicator in monetary I/O
- `showpoint` - generate a decimal-point character unconditionally for floating-point number output
- `showpos` - generate a `+` character for non-negative numeric output
- `skipws` - skip leading whitespace before certain input operations
- `unitbuf` flush the output after each output operation
- `uppercase` - replace certain lowercase letters with their uppercase equivalents in certain output output operations

Example of manipulators:

```
#include <iostream>
#include <string>
#include<iomanip>
int main()
{
  int l_iTemp = 47;
  std::cout<<  std::resetiosflags(std::ios_base::basefield);
  std::cout<<std::setiosflags( std::ios_base::oct)<<l_iTemp<<std::endl;
  //output: 57
  std::cout<<  std::resetiosflags(std::ios_base::basefield);
  std::cout<<std::setiosflags( std::ios_base::hex)<<l_iTemp<<std::endl;
  //output: 2f
  std::cout<<std::setiosflags( std::ios_base::uppercase)<<l_iTemp<<std::endl;
  //output 2F
  std::cout<<std::setfill('0')<<std::setw(12);
  std::cout<<std::resetiosflags(std::ios_base::uppercase);
  std::cout<<std::setiosflags( std::ios_base::right)<<l_iTemp<<std::endl;
  //output: 00000000002f

  std::cout<<std::resetiosflags(std::ios_base::basefield|std::ios_base::adjustfield);
  std::cout<<std::setfill('.')<<std::setw(10);
  std::cout<<std::setiosflags( std::ios_base::left)<<l_iTemp<<std::endl;
  //output: 47........

  std::cout<<std::resetiosflags(std::ios_base::adjustfield)<<std::setfill('#');
  std::cout<<std::setiosflags(std::ios_base::internal|std::ios_base::showpos);
  std::cout<<std::setw(10)<<l_iTemp<<std::endl;
  //output +#######47

  double l_dTemp = -1.2;
  double pi = 3.14159265359;
  std::cout<<pi<<"    "<<l_dTemp<<std::endl;
  //output +3.14159   -1.2
  std::cout<<std::setiosflags(std::ios_base::showpoint)<<l_dTemp<<std::endl;
  //output -1.20000
  std::cout<<setiosflags(std::ios_base::scientific)<<pi<<std::endl;
  //output: +3.141593e+00
```

```
    std::cout<<std::resetiosflags(std::ios_base::floatfield);
    std::cout<<setiosflags(std::ios_base::fixed)<<pi<<std::endl;
    //output: +3.141593
    bool b = true;
    std::cout<<std::setiosflags(std::ios_base::unitbuf|std::ios_base::boolalpha)<<b;
    //output: true
    return 0;
}
```

Section 57.4: std::setw

```
int val = 10;
// val will be printed to the extreme left end of the  output console:
std::cout << val << std::endl;
 // val will be printed in an output field of length 10 starting from right end of the field:
std::cout << std::setw(10) << val << std::endl;
```

This outputs:

```
10
        10
1234567890
```

(where the last line is there to aid in seeing the character offsets).

Sometimes we need to set the width of the output field, usually when we need to get the output in some structured and proper layout. That can be done using **`std::setw`** of **std::iomanip**.

The syntax for `std::setw` is:

```
std::setw(int n)
```

where n is the length of the output field to be set

Chapter 58: std::any

Section 58.1: Basic usage

```
std::any an_object{ std::string("hello world") };
if (an_object.has_value()) {
    std::cout << std::any_cast<std::string>(an_object) << '\n';
}

try {
  std::any_cast<int>(an_object);
} catch(std::bad_any_cast&) {
    std::cout << "Wrong type\n";
}

std::any_cast<std::string&>(an_object) = "42";
std::cout << std::any_cast<std::string>(an_object) << '\n';
```

Output

```
hello world
Wrong type
42
```

Chapter 59: std::set and std::multiset

`set` is a type of container whose elements are sorted and unique. `multiset` is similar, but, in the case of `multiset`, multiple elements can have the same value.

Section 59.1: Changing the default sort of a set

`set` and `multiset` have default compare methods, but in some cases you may need to overload them.

Let's imagine we are storing string values in a set, but we know those strings contain only numeric values. By default the sort will be a lexicographical string comparison, so the order won't match the numerical sort. If you want to apply a sort equivalent to what you would have with `int` values, you need a functor to overload the compare method:

```
#include <iostream>
#include <set>
#include <stdlib.h>

struct custom_compare final
{
    bool operator() (const std::string& left, const std::string& right) const
    {
        int nLeft = atoi(left.c_str());
        int nRight = atoi(right.c_str());
        return nLeft < nRight;
    }
};

int main ()
{
    std::set<std::string> sut({"1", "2", "5", "23", "6", "290"});

    std::cout << "### Default sort on std::set<std::string> :" << std::endl;
    for (auto &&data: sut)
        std::cout << data << std::endl;

    std::set<std::string, custom_compare> sut_custom({"1", "2", "5", "23", "6", "290"},
                                                     custom_compare{}); //< Compare object optional
as its default constructible.

    std::cout << std::endl << "### Custom sort on set :" << std::endl;
    for (auto &&data : sut_custom)
        std::cout << data << std::endl;

    auto compare_via_lambda = [](auto &&lhs, auto &&rhs){ return lhs > rhs; };
    using set_via_lambda = std::set<std::string, decltype(compare_via_lambda)>;
    set_via_lambda sut_reverse_via_lambda({"1", "2", "5", "23", "6", "290"},
                                          compare_via_lambda);

    std::cout << std::endl << "### Lambda sort on set :" << std::endl;
    for (auto &&data : sut_reverse_via_lambda)
        std::cout << data << std::endl;

    return 0;
}
```

Output will be:

```
### Default sort on std::set<std::string> :
1
2
23
290
5
6
### Custom sort on set :
1
2
5
6
23
290

### Lambda sort on set :
6
5
290
23
2
1
```

In the example above, one can find 3 different ways of adding compare operations to the `std::set`, each of them is useful in its own context.

Default sort

This will use the compare operator of the key (first template argument). Often, the key will already provide a good default for the `std::less<T>` function. Unless this function is specialized, it uses the `operator<` of the object. This is especially useful when other code also tries to use some ordering, as this allows consistency over the whole code base.

Writing the code this way, will reduce the effort to update your code when the key changes is API, like: a class containing 2 members which changes to a class containing 3 members. By updating the `operator<` in the class, all occurrences will get updated.

As you might expect, using the default sort is a reasonable default.

Custom sort

Adding a custom sort via an object with a compare operator is often used when the default comparison doesn't comply. In the example above this is because the strings are referring to integers. In other cases, it's often used when you want to compare (smart) pointers based upon the object they refer to or because you need different constraints for comparing (example: comparing `std::pair` by the value of `first`).

When creating a compare operator, this should be a stable sorting. If the result of the compare operator changes after insert, you will have undefined behavior. As a good practice, your compare operator should only use the constant data (const members, const functions ...).

As in the example above, you will often encounter classes without members as compare operators. This results in default constructors and copy constructors. The default constructor allows you to omit the instance at construction time and the copy constructor is required as the set takes a copy of the compare operator.

Lambda sort

Lambdas are a shorter way to write function objects. This allows writing the compare operator on less lines, making

the overall code more readable.

The disadvantage of the use of lambdas is that each lambda gets a specific type at compile time, so `decltype(lambda)` will be different for each compilation of the same compilation unit (cpp file) as over multiple compilation units (when included via header file). For this reason, its recommended to use function objects as compare operator when used within header files.

This construction is often encountered when a `std::set` is used within the local scope of a function instead, while the function object is preferred when used as function arguments or class members.

Other sort options

As the compare operator of `std::set` is a template argument, all callable objects can be used as compare operator and the examples above are only specific cases. The only restrictions these callable objects have are:

- They must be copy constructable
- They must be callable with 2 arguments of the type of the key. (implicit conversions are allowed, though not recommended as it can hurt performance)

Section 59.2: Deleting values from a set

The most obvious method, if you just want to reset your set/multiset to an empty one, is to use `clear`:

```
  std::set<int> sut;
  sut.insert(10);
  sut.insert(15);
  sut.insert(22);
  sut.insert(3);
  sut.clear(); //size of sut is 0
```

Then the `erase` method can be used. It offers some possibilities looking somewhat equivalent to the insertion:

```
std::set<int> sut;
std::set<int>::iterator it;

sut.insert(10);
sut.insert(15);
sut.insert(22);
sut.insert(3);
sut.insert(30);
sut.insert(33);
sut.insert(45);

// Basic deletion
sut.erase(3);

// Using iterator
it = sut.find(22);
sut.erase(it);

// Deleting a range of values
it = sut.find(33);
sut.erase(it, sut.end());

std::cout << std::endl << "Set under test contains:" << std::endl;
for (it = sut.begin(); it != sut.end(); ++it)
{
  std::cout << *it << std::endl;
```

```
}
```

Output will be:

```
Set under test contains:


10


15


30
```

All those methods also apply to `multiset`. Please note that if you ask to delete an element from a `multiset`, and it is present multiple times, **all the equivalent values will be deleted**.

Section 59.3: Inserting values in a set

Three different methods of insertion can used with sets.

- First, a simple insert of the value. This method returns a pair allowing the caller to check whether the insert really occurred.
- Second, an insert by giving a hint of where the value will be inserted. The objective is to optimize the insertion time in such a case, but knowing where a value should be inserted is not the common case. **Be careful in that case; the way to give a hint differs with compiler versions**.
- Finally you can insert a range of values by giving a starting and an ending pointer. The starting one will be included in the insertion, the ending one is excluded.

```
#include <iostream>
#include <set>

int main ()
{
  std::set<int> sut;
  std::set<int>::iterator it;
  std::pair<std::set<int>::iterator,bool> ret;

  // Basic insert
  sut.insert(7);
  sut.insert(5);
  sut.insert(12);

  ret = sut.insert(23);
  if (ret.second==true)
    std::cout << "# 23 has been inserted!" << std::endl;

  ret = sut.insert(23); // since it's a set and 23 is already present in it, this insert should
fail
  if (ret.second==false)
    std::cout << "# 23 already present in set!" << std::endl;

  // Insert with hint for optimization
  it = sut.end();
  // This case is optimized for C++11 and above
```

```
  // For earlier version, point to the element preceding your insertion
  sut.insert(it, 30);

  // inserting a range of values
  std::set<int> sut2;
  sut2.insert(20);
  sut2.insert(30);
  sut2.insert(45);
  std::set<int>::iterator itStart = sut2.begin();
  std::set<int>::iterator itEnd = sut2.end();

  sut.insert (itStart, itEnd); // second iterator is excluded from insertion

  std::cout << std::endl << "Set under test contains:" << std::endl;
  for (it = sut.begin(); it != sut.end(); ++it)
  {
    std::cout << *it << std::endl;
  }

  return 0;
}
```

Output will be:

```
# 23 has been inserted!

# 23 already present in set!

Set under test contains:

5

7

12

20

23

30
```

```
45
```

Section 59.4: Inserting values in a multiset

All the insertion methods from sets also apply to multisets. Nevertheless, another possibility exists, which is providing an initializer_list:

```
auto il = { 7, 5, 12 };
std::multiset<int> msut;
msut.insert(il);
```

Section 59.5: Searching values in set and multiset

There are several ways to search a given value in `std::set` or in `std::multiset`:

To get the iterator of the first occurrence of a key, the `find()` function can be used. It returns `end()` if the key does not exist.

```
  std::set<int> sut;
  sut.insert(10);
  sut.insert(15);
  sut.insert(22);
  sut.insert(3); // contains 3, 10, 15, 22

  auto itS = sut.find(10); // the value is found, so *itS == 10
  itS = sut.find(555); // the value is not found, so itS == sut.end()

  std::multiset<int> msut;
  sut.insert(10);
  sut.insert(15);
  sut.insert(22);
  sut.insert(15);
  sut.insert(3); // contains 3, 10, 15, 15, 22

  auto itMS = msut.find(10);
```

Another way is using the `count()` function, which counts how many corresponding values have been found in the `set/multiset` (in case of a `set`, the return value can be only 0 or 1). Using the same values as above, we will have:

```
int result = sut.count(10); // result == 1
result = sut.count(555); // result == 0

result = msut.count(10); // result == 1
result = msut.count(15); // result == 2
```

In the case of `std::multiset`, there could be several elements having the same value. To get this range, the `equal_range()` function can be used. It returns `std::pair` having iterator lower bound (inclusive) and upper bound (exclusive) respectively. If the key does not exist, both iterators would point to the nearest superior value (based on compare method used to sort the given `multiset`).

```
auto eqr = msut.equal_range(15);
auto st = eqr.first; // point to first element '15'
auto en = eqr.second; // point to element '22'
```

```
eqr = msut.equal_range(9); // both eqr.first and eqr.second point to element '10'
```

Chapter 60: std::integer_sequence

The class template `std::integer_sequence<Type, Values...>` represents a sequence of values of type `Type` where `Type` is one of the built-in integer types. These sequences are used when implementing class or function templates which benefit from positional access. The standard library also contains "factory" types which create ascending sequences of integer values just from the number of elements.

Section 60.1: Turn a std::tuple<T...> into function parameters

A `std::tuple<T...>` can be used to pass multiple values around. For example, it could be used to store a sequence of parameters into some form of a queue. When processing such a tuple its elements need to be turned into function call arguments:

```
#include <array>
#include <iostream>
#include <string>
#include <tuple>
#include <utility>

// ----------------------------------------------------------------------------
// Example functions to be called:
void f(int i, std::string const& s) {
    std::cout << "f(" << i << ", " << s << ")\n";
}
void f(int i, double d, std::string const& s) {
    std::cout << "f(" << i << ", " << d << ", " << s << ")\n";
}
void f(char c, int i, double d, std::string const& s) {
    std::cout << "f(" << c << ", " << i << ", " << d << ", " << s << ")\n";
}
void f(int i, int j, int k) {
    std::cout << "f(" << i << ", " << j << ", " << k << ")\n";
}

// ----------------------------------------------------------------------------
// The actual function expanding the tuple:
template <typename Tuple, std::size_t... I>
void process(Tuple const& tuple, std::index_sequence<I...>) {
    f(std::get<I>(tuple)...);
}

// The interface to call. Sadly, it needs to dispatch to another function
// to deduce the sequence of indices created from std::make_index_sequence<N>
template <typename Tuple>
void process(Tuple const& tuple) {
    process(tuple, std::make_index_sequence<std::tuple_size<Tuple>::value>());
}

// ----------------------------------------------------------------------------
int main() {
    process(std::make_tuple(1, 3.14, std::string("foo")));
    process(std::make_tuple('a', 2, 2.71, std::string("bar")));
    process(std::make_pair(3, std::string("pair")));
    process(std::array<int, 3>{ 1, 2, 3 });
}
```

As long as a class supports `std::get<I>(object)` and `std::tuple_size<T>::value` it can be expanded with the above `process()` function. The function itself is entirely independent of the number of arguments.

Section 60.2: Create a parameter pack consisting of integers

std::integer_sequence itself is about holding a sequence of integers which can be turned into a parameter pack. Its primary value is the possibility to create "factory" class templates creating these sequences:

```
#include <iostream>
#include <initializer_list>
#include <utility>

template <typename T, T... I>
void print_sequence(std::integer_sequence<T, I...>) {
    std::initializer_list<bool>{ bool(std::cout << I << ' ')... };
    std::cout << '\n';
}

template <int Offset, typename T, T... I>
void print_offset_sequence(std::integer_sequence<T, I...>) {
    print_sequence(std::integer_sequence<T, T(I + Offset)...>());
}

int main() {
    // explicitly specify sequences:
    print_sequence(std::integer_sequence<int, 1, 2, 3>());
    print_sequence(std::integer_sequence<char, 'f', 'o', 'o'>());

    // generate sequences:
    print_sequence(std::make_index_sequence<10>());
    print_sequence(std::make_integer_sequence<short, 10>());
    print_offset_sequence<'A'>(std::make_integer_sequence<char, 26>());
}
```

The print_sequence() function template uses an std::initializer_list<bool> when expanding the integer sequence to guarantee the order of evaluation and not creating an unused [array] variable.

Section 60.3: Turn a sequence of indices into copies of an element

Expanding the parameter pack of indices in a comma expression with a value creates a copy of the value for each of the indices. Sadly, gcc and clang think the index has no effect and warn about it (gcc can be silenced by casting the index to void):

```
#include <algorithm>
#include <array>
#include <iostream>
#include <iterator>
#include <string>
#include <utility>

template <typename T, std::size_t... I>
std::array<T, sizeof...(I)> make_array(T const& value, std::index_sequence<I...>) {
    return std::array<T, sizeof...(I)>{ (I, value)... };
}

template <int N, typename T>
std::array<T, N> make_array(T const& value) {
    return make_array(value, std::make_index_sequence<N>());
}

int main() {
```

```
    auto array = make_array<20>(std::string("value"));
    std::copy(array.begin(), array.end(),
              std::ostream_iterator<std::string>(std::cout, " "));
    std::cout << "\n";
}
```

Chapter 61: Using std::unordered_map

std::unordered_map is just an associative container. It works on keys and their maps. Key as the names goes, helps to have uniqueness in the map. While the mapped value is just a content that is associated with the key. The data types of this key and map can be any of the predefined data type or user-defined.

Section 61.1: Declaration and Usage

As already mentioned you can declare an unordered map of any type. Let's have a unordered map named first with string and integer type.

```
unordered_map<string, int> first; //declaration of the map
first["One"] = 1; // [] operator used to insert the value
first["Two"] = 2;
first["Three"] = 3;
first["Four"] = 4;
first["Five"] = 5;

pair <string,int> bar = make_pair("Nine", 9); //make a pair of same type
first.insert(bar); //can also use insert to feed the values
```

Section 61.2: Some Basic Functions

```
unordered_map<data_type, data_type> variable_name; //declaration

variable_name[key_value] = mapped_value; //inserting values

variable_name.find(key_value); //returns iterator to the key value

variable_name.begin(); // iterator to the first element

variable_name.end(); // iterator to the last + 1 element
```

Chapter 62: Standard Library Algorithms

Section 62.1: std::next_permutation

```
template< class Iterator >
bool next_permutation( Iterator first, Iterator last );
template< class Iterator, class Compare >
bool next_permutation( Iterator first, Iterator last, Compare cmpFun );
```

Effects:
Sift the data sequence of the range [first, last) into the next lexicographically higher permutation. If `cmpFun` is provided, the permutation rule is customized.

Parameters:
`first`- the beginning of the range to be permutated, inclusive
`last` - the end of the range to be permutated, exclusive

Return Value:
Returns true if such permutation exists.
Otherwise the range is swaped to the lexicographically smallest permutation and return false.

Complexity:
O(n), n is the distance from `first` to `last`.

Example:

```
std::vector< int > v { 1, 2, 3 };
do
{
   for( int i = 0; i < v.size(); i += 1 )
   {
       std::cout << v[i];
   }
   std::cout << std::endl;
}while( std::next_permutation( v.begin(), v.end() ) );
```

print all the permutation cases of 1,2,3 in lexicographically-increasing order.
output:

```
123
132
213
231
312
321
```

Section 62.2: std::for_each

```
template<class InputIterator, class Function>
    Function for_each(InputIterator first, InputIterator last, Function f);
```

Effects:

Applies `f` to the result of dereferencing every iterator in the range `[first, last)` starting from `first` and

proceeding to `last - 1`.

Parameters:

`first, last` - the range to apply `f` to.

`f` - callable object which is applied to the result of dereferencing every iterator in the range `[first, last)`.

Return value:

`f` (until C++11) and `std::move(f)` (since C++11).

Complexity:

Applies `f` exactly `last - first` times.

Example:

Version ≥ c++11

```
std::vector<int> v { 1, 2, 4, 8, 16 };
std::for_each(v.begin(), v.end(), [](int elem) { std::cout << elem << " "; });
```

Applies the given function for every element of the vector `v` printing this element to `stdout`.

Section 62.3: std::accumulate

Defined in header **`<numeric>`**

```
template<class InputIterator, class T>
T accumulate(InputIterator first, InputIterator last, T init); // (1)

template<class InputIterator, class T, class BinaryOperation>
T accumulate(InputIterator first, InputIterator last, T init, BinaryOperation f); // (2)
```

Effects:

std::accumulate performs fold operation using `f` function on range `[first, last)` starting with `init` as accumulator value.

Effectively it's equivalent of:

```
T acc = init;
for (auto it = first; first != last; ++it)
    acc = f(acc, *it);
return acc;
```

In version (1) `operator+` is used in place of `f`, so accumulate over container is equivalent of sum of container elements.

Parameters:

`first, last` - the range to apply `f` to.
`init` - initial value of accumulator.
`f` - binary folding function.

Return value:

Accumulated value of `f` applications.

Complexity:

O(n×k), where *n* is the distance from `first` to `last`, *O(k)* is complexity of `f` function.

Example:

Simple sum example:

```
std::vector<int> v { 2, 3, 4 };
auto sum = std::accumulate(v.begin(), v.end(), 1);
std::cout << sum << std::endl;
```

Output:

```
10
```

Convert digits to number:

Version < c++11

```
class Converter {
public:
    int operator()(int a, int d) const { return a * 10 + d; }
};
```

and later

```
const int ds[3] = {1, 2, 3};
int n = std::accumulate(ds, ds + 3, 0, Converter());
std::cout << n << std::endl;
```

Version ≥ c++11

```
const std::vector<int> ds = {1, 2, 3};
int n = std::accumulate(ds.begin(), ds.end(),
                        0,
                        [](int a, int d) { return a * 10 + d; });
std::cout << n << std::endl;
```

Output:

```
123
```

Section 62.4: std::find

```
template <class InputIterator, class T>
InputIterator find (InputIterator first, InputIterator last, const T& val);
```

Effects

Finds the first occurrence of val within the range [first, last)

Parameters

`first` => iterator pointing to the beginning of the range `last` => iterator pointing to the end of the range `val` => The value to find within the range

Return

An iterator that points to the first element within the range that is equal(==) to val, the iterator points to last if val is not found.

Example

```
#include <vector>
#include <algorithm>
#include <iostream>

using namespace std;

int main(int argc, const char * argv[]) {

  //create a vector
  vector<int> intVec {4, 6, 8, 9, 10, 30, 55,100, 45, 2, 4, 7, 9, 43, 48};

  //define iterators
  vector<int>::iterator  itr_9;
  vector<int>::iterator  itr_43;
  vector<int>::iterator  itr_50;

  //calling find
  itr_9 = find(intVec.begin(), intVec.end(), 9); //occurs twice
  itr_43 = find(intVec.begin(), intVec.end(), 43); //occurs once

  //a value not in the vector
  itr_50 = find(intVec.begin(), intVec.end(), 50); //does not occur

  cout << "first occurrence of: " << *itr_9 << endl;
  cout << "only occurrence of: " << *itr_43 << Lendl;


  /*
    let's prove that itr_9 is pointing to the first occurrence
    of 9 by looking at the element after 9, which should be 10
    not 43
  */
  cout << "element after first 9: " << *(itr_9 + 1) << ends;

  /*
    to avoid dereferencing intVec.end(), lets look at the
    element right before the end
  */
  cout << "last element: " << *(itr_50 - 1) << endl;

  return 0;
}
```

Output

```
first occurrence of: 9
only occurrence of: 43
element after first 9: 10
last element: 48
```

Section 62.5: std::min_element

```
template <class ForwardIterator>
ForwardIterator min_element (ForwardIterator first, ForwardIterator last);

template <class ForwardIterator, class Compare>
ForwardIterator min_element (ForwardIterator first, ForwardIterator last,Compare comp);
```

Effects

Finds the minimum element in a range

Parameters

`first` - iterator pointing to the beginning of the range
`last` - iterator pointing to the end of the range `comp` - a function pointer or function object that takes two arguments and returns true or false indicating whether argument is less than argument 2. This function should not modify inputs

Return

Iterator to the minimum element in the range

Complexity

Linear in one less than the number of elements compared.

Example

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <utility>  //to use make_pair

using namespace std;

//function compare two pairs
bool pairLessThanFunction(const pair<string, int> &p1, const pair<string, int> &p2)
{
  return p1.second < p2.second;
}

int main(int argc, const char * argv[]) {

  vector<int> intVec {30,200,167,56,75,94,10,73,52,6,39,43};

  vector<pair<string, int>> pairVector = {make_pair("y", 25), make_pair("b", 2), make_pair("z",
26), make_pair("e", 5) };


  // default using < operator
  auto minInt = min_element(intVec.begin(), intVec.end());

  //Using pairLessThanFunction
  auto minPairFunction = min_element(pairVector.begin(), pairVector.end(), pairLessThanFunction);


  //print minimum of intVector
  cout << "min int from default: " << *minInt << endl;
```

```
  //print minimum of pairVector
  cout << "min pair from PairLessThanFunction: " << (*minPairFunction).second << endl;

  return 0;
}
```

Output

```
min int from default: 6
min pair from PairLessThanFunction: 2
```

Section 62.6: std::find_if

```
template <class InputIterator, class UnaryPredicate>
InputIterator find_if (InputIterator first, InputIterator last, UnaryPredicate pred);
```

Effects

Finds the first element in a range for which the predicate function `pred` returns true.

Parameters

`first` => iterator pointing to the beginning of the range `last` => iterator pointing to the end of the range `pred` => predicate function(returns true or false)

Return

An iterator that points to the first element within the range the predicate function pred returns true for. The iterator points to last if val is not found

Example

```
#include <iostream>
#include <vector>
#include <algorithm>


using namespace std;

/*
    define some functions to use as predicates
*/

//Returns true if x is multiple of 10
bool multOf10(int x) {
  return x % 10 == 0;
}

//returns true if item greater than passed in parameter
class Greater {
  int _than;

public:
  Greater(int th):_than(th){

  }
  bool operator()(int data) const
```

```
  {
    return data > _than;
  }
};


int main()
{

  vector<int> myvec {2, 5, 6, 10, 56, 7, 48, 89, 850, 7, 456};

  //with a lambda function
  vector<int>::iterator gt10 = find_if(myvec.begin(), myvec.end(), [](int x){return x>10;}); // >=
C++11

  //with a function pointer
  vector<int>::iterator pow10 = find_if(myvec.begin(), myvec.end(), multOf10);

  //with functor
  vector<int>::iterator gt5 = find_if(myvec.begin(), myvec.end(), Greater(5));

  //not Found
  vector<int>::iterator nf = find_if(myvec.begin(), myvec.end(), Greater(1000)); // nf points to
myvec.end()


  //check if pointer points to myvec.end()
  if(nf != myvec.end()) {
    cout << "nf points to: " << *nf << endl;
  }
  else {
    cout << "item not found" << endl;
  }



  cout << "First item >   10: " << *gt10  << endl;
  cout << "First Item n * 10: " << *pow10 << endl;
  cout << "First Item >    5: " << *gt5   << endl;

  return 0;
}
```

Output

```
item not found
First item >   10: 56
First Item n * 10: 10
First Item >    5: 6
```

Section 62.7: Using std::nth_element To Find The Median (Or Other Quantiles)

The `std::nth_element` algorithm takes three iterators: an iterator to the beginning, *n*th position, and end. Once the function returns, the *n*th element (by order) will be the *n*th smallest element. (The function has more elaborate overloads, e.g., some taking comparison functors; see the above link for all the variations.)

Note This function is very efficient - it has linear complexity.

For the sake of this example, let's define the median of a sequence of length *n* as the element that would be in position ⌈n / 2⌉. For example, the median of a sequence of length 5 is the 3rd smallest element, and so is the median of a sequence of length 6.

To use this function to find the median, we can use the following. Say we start with

```
std::vector<int> v{5, 1, 2, 3, 4};

std::vector<int>::iterator b = v.begin();
std::vector<int>::iterator e = v.end();

std::vector<int>::iterator med = b;
std::advance(med, v.size() / 2);

// This makes the 2nd position hold the median.
std::nth_element(b, med, e);

// The median is now at v[2].
```

To find the *p*th quantile, we would change some of the lines above:

```
const std::size_t pos = p * std::distance(b, e);

std::advance(nth, pos);
```

and look for the quantile at position `pos`.

Section 62.8: std::count

```
template <class InputIterator, class T>
typename iterator_traits<InputIterator>::difference_type
count (InputIterator first, InputIterator last, const T& val);
```

Effects

Counts the number of elements that are equal to val

Parameters

`first` => iterator pointing to the beginning of the range
`last` => iterator pointing to the end of the range
`val` => The occurrence of this value in the range will be counted

Return

The number of elements in the range that are equal(==) to val.

Example

```
#include <vector>
#include <algorithm>
#include <iostream>

using namespace std;

int main(int argc, const char * argv[]) {
```

```
    //create vector
    vector<int> intVec{4,6,8,9,10,30,55,100,45,2,4,7,9,43,48};

    //count occurrences of 9, 55, and 101
    size_t count_9 = count(intVec.begin(), intVec.end(), 9); //occurs twice
    size_t count_55 = count(intVec.begin(), intVec.end(), 55); //occurs once
    size_t count_101 = count(intVec.begin(), intVec.end(), 101); //occurs once

    //print result
    cout << "There are " << count_9  << " 9s"<< endl;
    cout << "There is " << count_55  << " 55"<< endl;
    cout << "There is " << count_101  << " 101"<< ends;

    //find the first element == 4 in the vector
    vector<int>::iterator itr_4 = find(intVec.begin(), intVec.end(), 4);

    //count its occurrences in the vector starting from the first one
    size_t count_4 = count(itr_4, intVec.end(), *itr_4); // should be 2

    cout << "There are " << count_4  << " " << *itr_4 << endl;

    return 0;
}
```

Output

```
There are 2 9s
There is 1 55
There is 0 101
There are 2 4
```

Section 62.9: std::count_if

```
template <class InputIterator, class UnaryPredicate>
typename iterator_traits<InputIterator>::difference_type
count_if (InputIterator first, InputIterator last, UnaryPredicate red);
```

Effects

Counts the number of elements in a range for which a specified predicate function is true

Parameters

`first` => iterator pointing to the beginning of the range `last` => iterator pointing to the end of the range `red` => predicate function(returns true or false)

Return

The number of elements within the specified range for which the predicate function returned true.

Example

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;
```

```
/*
    Define a few functions to use as predicates
*/

//return true if number is odd
bool isOdd(int i){
  return i%2 == 1;
}

//functor that returns true if number is greater than the value of the constructor parameter
provided
class Greater {
  int _than;
public:
  Greater(int th): _than(th){}
  bool operator()(int i){
    return i > _than;
  }
};

int main(int argc, const char * argv[]) {

  //create a vector
  vector<int> myvec = {1,5,8,0,7,6,4,5,2,1,5,0,6,9,7};

  //using a lambda function to count even numbers
  size_t evenCount = count_if(myvec.begin(), myvec.end(), [](int i){return i % 2 == 0;}); // >=
C++11

  //using function pointer to count odd number in the first half of the vector
  size_t oddCount = count_if(myvec.begin(), myvec.end()- myvec.size()/2, isOdd);

  //using a functor to count numbers greater than 5
  size_t greaterCount = count_if(myvec.begin(), myvec.end(), Greater(5));

  cout << "vector size: " << myvec.size() << endl;
  cout << "even numbers: " << evenCount << " found" << endl;
  cout << "odd numbers: " << oddCount << " found" << endl;
  cout << "numbers > 5: " << greaterCount << " found"<< endl;

  return 0;
}
```

Output

```
vector size: 15
even numbers: 7 found
odd numbers: 4 found
numbers > 5: 6 found
```

Chapter 63: The ISO C++ Standard

In 1998, the there was a first publication of the standard making C++ an internally standardized language. From that time, C++ has evolved resulting in different dialects of C++. On this page, you can find an overview of all different standards and their changes compared to the previous version. The details on how to use these features is described on more specialized pages.

Section 63.1: Current Working Drafts

All published ISO standards are available for sale from the ISO store (http://www.iso.org). The working drafts of the C++ standards are publicly available for free though.

The different versions of the standard:

- Upcoming (Sometimes referred as C++20 or C++2a): Current working draft (HTML-version)
- Proposed (Sometimes referred as C++17 or C++1z): March 2017 working draft N4659.
- C++14 (Sometimes referred as C++1y): November 2014 working draft N4296
- C++11 (Sometimes referred as C++0x): February 2011 working draft N3242
- C++03
- C++98

Section 63.2: C++17

The C++17 standard is feature complete and has been proposed for standardization. In compilers with experimental support for these features, it is usually referred to as C++1z.

Language Extensions

- Fold Expressions
- declaring non-type template arguments with `auto`
- Guaranteed copy elision
- Template parameter deduction for constructors
- Structured bindings
- Compact nested namespaces
- New attributes: `[[fallthrough]]`, `[[nodiscard]]`, `[[maybe_unused]]`
- Default message for `static_assert`
- Initializers in `if` and `switch`
- Inline variables
- `if constexpr`
- Order of expression evaluation guarantees
- Dynamic memory allocation for over-aligned data

Library Extensions

- `std::optional`
- `std::variant`
- `std::string_view`
- `merge()` and `extract()` for associative containers
- A file system library with the **`<filesystem>`** header.
- Parallel versions of most of the standard algorithms (in the **`<algorithm>`** header).
- Addition of mathematical special functions in the **`<cmath>`** header.
- Moving nodes between map<>, unordered_map<>, set<>, and unordered_set<>

Section 63.3: C++11

The C++11 standard is a major extension to the C++ standard. Below you can find an overview of the changes as they have been grouped on the isocpp FAQ with links to more detailed documentation.

Language Extensions

General Features

- auto
- decltype
- Range-for statement
- Initializer lists
- Uniform initialization syntax and semantics
- Rvalue references and move semantics
- Lambdas
- noexcept to prevent exception propagation
- constexpr
- nullptr – a null pointer literal
- Copying and rethrowing exceptions
- Inline namespaces
- User-defined literals

Classes

- =default and =delete
- Control of default move and copy
- Delegating constructors
- In-class member initializers
- Inherited constructors
- Override controls: override
- Override controls: final
- Explicit conversion operators

Other Types

- enum class
- long long – a longer integer
- Extended integer types
- Generalized unions
- Generalized PODs

Templates

- Extern templates
- Template aliases
- Variadic templates
- Local types as template arguments

Concurrency

- Concurrency memory model
- Dynamic initialization and destruction with concurrency
- Thread-local storage

Miscellaneous Language Features

- What is the value of __cplusplus for C++11?
- Suffix return type syntax
- Preventing narrowing
- Right-angle brackets
- static_assert compile-time assertions
- Raw string literals
- Attributes
- Alignment
- C99 features

Library Extensions
General

- unique_ptr
- shared_ptr
- weak_ptr
- Garbage collection ABI
- tuple
- Type traits
- function and bind
- Regular Expressions
- Time utilities
- Random number generation
- Scoped allocators

Containers and Algorithms

- Algorithms improvements
- Container improvements
- unordered_* containers
- std::array
- forward_list

Concurrency

- Threads
- Mutual exclusion
- Locks
- Condition variables
- Atomics
- Futures and promises
- async
- Abandoning a process

Section 63.4: C++14

The C++14 standard is often referred to as a bugfix for C++11. It contains only a limited list of changes of which most are extensions to the new features in C++11. Below you can find an overview of the changes as they have been grouped on the isocpp FAQ with links to more detailed documentation.

Language Extensions

- Binary literals
- Generalized return type deduction

- decltype(auto)
- Generalized lambda captures
- Generic lambdas
- Variable templates
- Extended `constexpr`
- The `[[deprecated]]` attribute
- Digit separators

Library Extensions

- Shared locking
- User-defined literals for `std::` types
- `std::make_unique`
- Type transformation `_t` aliases
- Addressing tuples by type (e.g. `get<string>(t)`)
- Transparent Operator Functors (e.g. `greater<>(x)`)
- `std::quoted`

Deprecated / Removed

- `std::gets` was deprecated in C++11 and removed from C++14
- `std::random_shuffle` is deprecated

Section 63.5: C++98

C++98 is the first standardized version of C++. As it was developed as an extension to C, many of the features which set apart C++ from C are added.

Language Extensions (in respect to C89/C90)

- Classes, Derived classes, virtual member functions, const member functions
- Function overloading, Operator overloading
- Single line comments (Has been introduced in the C-languague with C99 standard)
- References
- new and delete
- boolean type (Has been introduced in the C-languague with C99 standard)
- templates
- namespaces
- exceptions
- specific casts

Library Extensions

- The Standard Template Library

Section 63.6: C++03

The C++03 standard mainly addresses defect reports of the C++98 standard. Apart from these defects, it only adds one new feature.

Language Extensions

- Value initalization

Section 63.7: C++20

C++20 is the upcoming standard of C++, currently in development, based upon the C++17 standard. It's progress can be tracked on the official ISO cpp website.

The following features are simply what has been accepted for the next release of the C++ standard, targeted for 2020.

Language Extensions

No language extensions have been accepted for now.

Library Extensions

No library extensions have been accepted for now.

Chapter 64: Inline variables

An inline variable is allowed to be defined in multiple translation units without violating the One Definition Rule. If it is multiply defined, the linker will merge all definitions into a single object in the final program.

Section 64.1: Defining a static data member in the class definition

A static data member of the class may be fully defined within the class definition if it is declared `inline`. For example, the following class may be defined in a header. Prior to C++17, it would have been necessary to provide a `.cpp` file to contain the definition of `Foo::num_instances` so that it would be defined only once, but in C++17 the multiple definitions of the inline variable `Foo::num_instances` all refer to the same `int` object.

```
// warning: not thread-safe...
class Foo {
  public:
    Foo() { ++num_instances; }
    ~Foo() { --num_instances; }
    inline static int num_instances = 0;
};
```

As a special case, a `constexpr` static data member is implicitly inline.

```
class MyString {
  public:
    MyString() { /* ... */ }
    // ...
    static constexpr int max_size = INT_MAX / 2;
};
// in C++14, this definition was required in a single translation unit:
// constexpr int MyString::max_size;
```

Chapter 65: Random number generation

Section 65.1: True random value generator

To generate true random values that can be used for cryptography `std::random_device` has to be used as generator.

```
#include <iostream>
#include <random>

int main()
{
   std::random_device crypto_random_generator;
   std::uniform_int_distribution<int> int_distribution(0,9);

   int actual_distribution[10] = {0,0,0,0,0,0,0,0,0,0};

   for(int i = 0; i < 10000; i++) {
       int result = int_distribution(crypto_random_generator);
       actual_distribution[result]++;
   }

   for(int i = 0; i < 10; i++) {
       std::cout << actual_distribution[i] << " ";
   }

   return 0;
}
```

`std::random_device` is used in the same way as a pseudo random value generator is used.

However `std::random_device` **may be implemented in terms of an implementation-defined pseudo-random number engine** if a non-deterministic source (e.g. a hardware device) isn't available to the implementation.

Detecting such implementations should be possible via the `entropy` member function (which return zero when the generator is completely deterministic), but many popular libraries (both GCC's libstdc++ and LLVM's libc++) always return zero, even when they're using high-quality external randomness.

Section 65.2: Generating a pseudo-random number

A pseudo-random number generator generates values that can be guessed based on previously generated values. In other words: it is deterministic. Do not use a pseudo-random number generator in situations where a true random number is required.

```
#include <iostream>
#include <random>

int main()
{
   std::default_random_engine pseudo_random_generator;
   std::uniform_int_distribution<int> int_distribution(0, 9);

   int actual_distribution[10] = {0,0,0,0,0,0,0,0,0,0};

   for(int i = 0; i < 10000; i++) {
       int result = int_distribution(pseudo_random_generator);
       actual_distribution[result]++;
```

```
    }

    for(int i = 0; i <= 9; i++) {
        std::cout << actual_distribution[i] << " ";
    }

    return 0;
}
```

This code creates a random number generator, and a distribution that generates integers in the range [0,9] with equal likelihood. It then counts how many times each result was generated.

The template parameter of `std::uniform_int_distribution<T>` specifies the type of integer that should be generated. Use `std::uniform_real_distribution<T>` to generate floats or doubles.

Section 65.3: Using the generator for multiple distributions

The random number generator can (and should) be used for multiple distributions.

```
#include <iostream>
#include <random>

int main()
{
    std::default_random_engine pseudo_random_generator;
    std::uniform_int_distribution<int> int_distribution(0, 9);
    std::uniform_real_distribution<float> float_distribution(0.0, 1.0);
    std::discrete_distribution<int> rigged_dice({1,1,1,1,1,100});

    std::cout << int_distribution(pseudo_random_generator) << std::endl;
    std::cout << float_distribution(pseudo_random_generator) << std::endl;
    std::cout << (rigged_dice(pseudo_random_generator) + 1) << std::endl;

    return 0;
}
```

In this example, only one generator is defined. It is subsequently used to generate a random value in three different distributions. The `rigged_dice` distribution will generate a value between 0 and 5, but almost always generates a `5`, because the chance to generate a `5` is `100 / 105`.

Chapter 66: Date and time using <chrono> header

Section 66.1: Measuring time using <chrono>

The `system_clock` can be used to measure the time elapsed during some part of a program's execution.

Version = c++11

```
#include <iostream>
#include <chrono>
#include <thread>

int main() {
    auto start = std::chrono::system_clock::now(); // This and "end"'s type is
std::chrono::time_point
    { // The code to test
        std::this_thread::sleep_for(std::chrono::seconds(2));
    }
    auto end = std::chrono::system_clock::now();

    std::chrono::duration<double> elapsed = end - start;
    std::cout << "Elapsed time: " << elapsed.count() << "s";
}
```

In this example, `sleep_for` was used to make the active thread sleep for a time period measured in `std::chrono::seconds`, but the code between braces could be any function call that takes some time to execute.

Section 66.2: Find number of days between two dates

This example shows how to find number of days between two dates. A date is specified by year/month/day of month, and additionally hour/minute/second.

Program calculates number of days in years since 2000.

```
#include <iostream>
#include <string>
#include <chrono>
#include <ctime>

/***
 * Creates a std::tm structure from raw date.
 *
 * \param year (must be 1900 or greater)
 * \param month months since January – [1, 12]
 * \param day day of the month – [1, 31]
 * \param minutes minutes after the hour – [0, 59]
 * \param seconds seconds after the minute – [0, 61](until C++11) / [0, 60] (since C++11)
 *
 * Based on http://en.cppreference.com/w/cpp/chrono/c/tm
 */
std::tm CreateTmStruct(int year, int month, int day, int hour, int minutes, int seconds) {
    struct tm tm_ret = {0};

    tm_ret.tm_sec = seconds;
    tm_ret.tm_min = minutes;
    tm_ret.tm_hour = hour;
    tm_ret.tm_mday = day;
```

```
    tm_ret.tm_mon = month - 1;
    tm_ret.tm_year = year - 1900;

    return tm_ret;
}

int get_days_in_year(int year) {

    using namespace std;
    using namespace std::chrono;

    // We want results to be in days
    typedef duration<int, ratio_multiply<hours::period, ratio<24> >::type> days;

    // Create start time span
    std::tm tm_start = CreateTmStruct(year, 1, 1, 0, 0, 0);
    auto tms = system_clock::from_time_t(std::mktime(&tm_start));

    // Create end time span
        std::tm tm_end =   CreateTmStruct(year + 1, 1, 1, 0, 0, 0);
    auto tme = system_clock::from_time_t(std::mktime(&tm_end));

    // Calculate time duration between those two dates
    auto diff_in_days = std::chrono::duration_cast<days>(tme - tms);

    return diff_in_days.count();
}

int main()
{
    for ( int year = 2000; year <= 2016; ++year )
        std::cout << "There are " << get_days_in_year(year) << " days in " << year << "\n";
}
```

Chapter 67: Sorting

Section 67.1: Sorting and sequence containers

`std::sort`, found in the standard library header `algorithm`, is a standard library algorithm for sorting a range of values, defined by a pair of iterators. `std::sort` takes as the last parameter a functor used to compare two values; this is how it determines the order. Note that `std::sort` is not stable.

The comparison function *must* impose a Strict, Weak Ordering on the elements. A simple less-than (or greater-than) comparison will suffice.

A container with random-access iterators can be sorted using the `std::sort` algorithm:

Version ≥ C++11

```
#include <vector>
#include <algorithm>

std::vector<int> MyVector = {3, 1, 2}

//Default comparison of <
std::sort(MyVector.begin(), MyVector.end());
```

`std::sort` requires that its iterators are random access iterators. The sequence containers `std::list` and `std::forward_list` (requiring C++11) do not provide random access iterators, so they cannot be used with `std::sort`. However, they do have `sort` member functions which implement a sorting algorithm that works with their own iterator types.

Version ≥ C++11

```
#include <list>
#include <algorithm>

std::list<int> MyList = {3, 1, 2}

//Default comparison of <
//Whole list only.
MyList.sort();
```

Their member `sort` functions always sort the entire list, so they cannot sort a sub-range of elements. However, since `list` and `forward_list` have fast splicing operations, you could extract the elements to be sorted from the list, sort them, then stuff them back where they were quite efficiently like this:

```
void sort_sublist(std::list<int>& mylist, std::list<int>::const_iterator start,
std::list<int>::const_iterator end) {
    //extract and sort half-open sub range denoted by start and end iterator
    std::list<int> tmp;
    tmp.splice(tmp.begin(), list, start, end);
    tmp.sort();
    //re-insert range at the point we extracted it from
    list.splice(end, tmp);
}
```

Section 67.2: sorting with std::map (ascending and descending)

This example sorts elements in **ascending** order of a **key** using a map. You can use any type, including class,

instead of `std::string`, in the example below.

```
#include <iostream>
#include <utility>
#include <map>

int main()
{
    std::map<double, std::string> sorted_map;
    // Sort the names of the planets according to their size
    sorted_map.insert(std::make_pair(0.3829, "Mercury"));
    sorted_map.insert(std::make_pair(0.9499, "Venus"));
    sorted_map.insert(std::make_pair(1,      "Earth"));
    sorted_map.insert(std::make_pair(0.532,  "Mars"));
    sorted_map.insert(std::make_pair(10.97,  "Jupiter"));
    sorted_map.insert(std::make_pair(9.14,   "Saturn"));
    sorted_map.insert(std::make_pair(3.981,  "Uranus"));
    sorted_map.insert(std::make_pair(3.865,  "Neptune"));

    for (auto const& entry: sorted_map)
    {
        std::cout << entry.second << " (" << entry.first << " of Earth's radius)" << '\n';
    }
}
```

Output:

```
Mercury (0.3829 of Earth's radius)
Mars (0.532 of Earth's radius)
Venus (0.9499 of Earth's radius)
Earth (1 of Earth's radius)
Neptune (3.865 of Earth's radius)
Uranus (3.981 of Earth's radius)
Saturn (9.14 of Earth's radius)
Jupiter (10.97 of Earth's radius)
```

If entries with equal keys are possible, use `multimap` instead of `map` (like in the following example).

To sort elements in **descending** manner, declare the map with a proper comparison functor (`std::greater<>`):

```
#include <iostream>
#include <utility>
#include <map>

int main()
{
    std::multimap<int, std::string, std::greater<int>> sorted_map;
    // Sort the names of animals in descending order of the number of legs
    sorted_map.insert(std::make_pair(6,   "bug"));
    sorted_map.insert(std::make_pair(4,   "cat"));
    sorted_map.insert(std::make_pair(100, "centipede"));
    sorted_map.insert(std::make_pair(2,   "chicken"));
    sorted_map.insert(std::make_pair(0,   "fish"));
    sorted_map.insert(std::make_pair(4,   "horse"));
    sorted_map.insert(std::make_pair(8,   "spider"));

    for (auto const& entry: sorted_map)
    {
        std::cout << entry.second << " (has " << entry.first << " legs)" << '\n';
```

```
    }
}
```

Output

```
centipede (has 100 legs)
spider (has 8 legs)
bug (has 6 legs)
cat (has 4 legs)
horse (has 4 legs)
chicken (has 2 legs)
fish (has 0 legs)
```

Section 67.3: Sorting sequence containers by overloaded less operator

If no ordering function is passed, `std::sort` will order the elements by calling `operator<` on pairs of elements, which must return a type contextually convertible to `bool` (or just `bool`). Basic types (integers, floats, pointers etc) have already build in comparison operators.

We can overload this operator to make the default `sort` call work on user-defined types.

```
// Include sequence containers
#include <vector>
#include <deque>
#include <list>

// Insert sorting algorithm
#include <algorithm>

class Base {
 public:

    // Constructor that set variable to the value of v
    Base(int v): variable(v) {
    }

    // Use variable to provide total order operator less
    //`this` always represents the left-hand side of the compare.
    bool operator<(const Base &b) const {
        return this->variable < b.variable;
    }

    int variable;
};

int main() {
    std::vector <Base> vector;
    std::deque <Base> deque;
    std::list <Base> list;

    // Create 2 elements to sort
    Base a(10);
    Base b(5);

    // Insert them into backs of containers
    vector.push_back(a);
    vector.push_back(b);
```

```
    deque.push_back(a);
    deque.push_back(b);

    list.push_back(a);
    list.push_back(b);

    // Now sort data using operator<(const Base &b) function
    std::sort(vector.begin(), vector.end());
    std::sort(deque.begin(), deque.end());
    // List must be sorted differently due to its design
    list.sort();

    return 0;
}
```

Section 67.4: Sorting sequence containers using compare function

```
// Include sequence containers
#include <vector>
#include <deque>
#include <list>

// Insert sorting algorithm
#include <algorithm>

class Base {
 public:

    // Constructor that set variable to the value of v
    Base(int v): variable(v) {
    }

    int variable;
};

bool compare(const Base &a, const Base &b) {
    return a.variable < b.variable;
}

int main() {
    std::vector <Base> vector;
    std::deque <Base> deque;
    std::list <Base> list;

    // Create 2 elements to sort
    Base a(10);
    Base b(5);

    // Insert them into backs of containers
    vector.push_back(a);
    vector.push_back(b);

    deque.push_back(a);
    deque.push_back(b);

    list.push_back(a);
    list.push_back(b);

    // Now sort data using comparing function
```

```
    std::sort(vector.begin(), vector.end(), compare);
    std::sort(deque.begin(), deque.end(), compare);
    list.sort(compare);

    return 0;
}
```

Section 67.5: Sorting sequence containers using lambda expressions (C++11)

Version ≥ C++11

```
// Include sequence containers
#include <vector>
#include <deque>
#include <list>
#include <array>
#include <forward_list>

// Include sorting algorithm
#include <algorithm>

class Base {
 public:

    // Constructor that set variable to the value of v
    Base(int v): variable(v) {
    }

    int variable;
};


int main() {
    // Create 2 elements to sort
    Base a(10);
    Base b(5);

    // We're using C++11, so let's use initializer lists to insert items.
    std::vector <Base> vector = {a, b};
    std::deque <Base> deque = {a, b};
    std::list <Base> list = {a, b};
    std::array <Base, 2> array = {a, b};
    std::forward_list<Base> flist = {a, b};

    // We can sort data using an inline lambda expression
    std::sort(std::begin(vector), std::end(vector),
      [](const Base &a, const Base &b) { return a.variable < b.variable;});

    // We can also pass a lambda object as the comparator
    // and reuse the lambda multiple times
    auto compare = [](const Base &a, const Base &b) {
                     return a.variable < b.variable;};
    std::sort(std::begin(deque), std::end(deque), compare);
    std::sort(std::begin(array), std::end(array), compare);
    list.sort(compare);
    flist.sort(compare);

    return 0;
}
```

Section 67.6: Sorting built-in arrays

The `sort` algorithm sorts a sequence defined by two iterators. This is enough to sort a built-in (also known as c-style) array.

Version ≥ C++11

```
int arr1[] = {36, 24, 42, 60, 59};

// sort numbers in ascending order
sort(std::begin(arr1), std::end(arr1));

// sort numbers in descending order
sort(std::begin(arr1), std::end(arr1), std::greater<int>());
```

Prior to C++11, end of array had to be "calculated" using the size of the array:

Version < C++11

```
// Use a hard-coded number for array size
sort(arr1, arr1 + 5);

// Alternatively, use an expression
const size_t arr1_size = sizeof(arr1) / sizeof(*arr1);
sort(arr1, arr1 + arr1_size);
```

Section 67.7: Sorting sequence containers with specifed ordering

If the values in a container have certain operators already overloaded, `std::sort` can be used with specialized functors to sort in either ascending or descending order:

Version ≥ C++11

```
#include <vector>
#include <algorithm>
#include <functional>

std::vector<int> v = {5,1,2,4,3};

//sort in ascending order (1,2,3,4,5)
std::sort(v.begin(), v.end(), std::less<int>());

// Or just:
std::sort(v.begin(), v.end());

//sort in descending order (5,4,3,2,1)
std::sort(v.begin(), v.end(), std::greater<int>());

//Or just:
std::sort(v.rbegin(), v.rend());
```

Version ≥ C++14

In C++14, we don't need to provide the template argument for the comparison function objects and instead let the object deduce based on what it gets passed in:

```
std::sort(v.begin(), v.end(), std::less<>());     // ascending order
std::sort(v.begin(), v.end(), std::greater<>());  // descending order
```

Chapter 68: Enumeration

Section 68.1: Iteration over an enum

There is no built-in to iterate over enumeration.

But there are several ways

- for `enum` with only consecutive values:

```
enum E {
    Begin,
    E1 = Begin,
    E2,
    // ..
    En,
    End
};

for (E e = E::Begin; e != E::End; ++e) {
    // Do job with e
}
```

Version ≥ C++11

with `enum class`, `operator ++` has to be implemented:

```
E& operator ++ (E& e)
{
    if (e == E::End) {
        throw std::out_of_range("for E& operator ++ (E&)");
    }
    e = E(static_cast<std::underlying_type<E>::type>(e) + 1);
    return e;
}
```

- using a container as `std::vector`

```
enum E {
    E1 = 4,
    E2 = 8,
    // ..
    En
};

std::vector<E> build_all_E()
{
    const E all[] = {E1, E2, /*..*/ En};

    return std::vector<E>(all, all + sizeof(all) / sizeof(E));
}

std::vector<E> all_E = build_all_E();
```

and then

```
for (std::vector<E>::const_iterator it = all_E.begin(); it != all_E.end(); ++it) {
```

```
    E e = *it;
    // Do job with e;
}
```

Version ≥ C++11

- or `std::initializer_list` and a simpler syntax:

```
enum E {
    E1 = 4,
    E2 = 8,
    // ..
    En
};

constexpr std::initializer_list<E> all_E = {E1, E2, /*..*/ En};
```

and then

```
for (auto e : all_E) {
    // Do job with e
}
```

Section 68.2: Scoped enums

C++11 introduces what are known as *scoped enums*. These are enumerations whose members must be qualified with `enumname::membername`. Scoped enums are declared using the `enum class` syntax. For example, to store the colors in a rainbow:

```
enum class rainbow {
    RED,
    ORANGE,
    YELLOW,
    GREEN,
    BLUE,
    INDIGO,
    VIOLET
};
```

To access a specific color:

```
rainbow r = rainbow::INDIGO;
```

`enum class`es cannot be implicitly converted to `int`s without a cast. So `int x = rainbow::RED` is invalid.

Scoped enums also allow you to specify the *underlying type*, which is the type used to represent a member. By default it is `int`. In a Tic-Tac-Toe game, you may store the piece as

```
enum class piece : char {
    EMPTY = '\0',
    X = 'X',
    O = 'O',
};
```

As you may notice, `enum`s can have a trailing comma after the last member.

Section 68.3: Enum forward declaration in C++11

Scoped enumerations:

```
...
enum class Status; // Forward declaration
Status doWork(); // Use the forward declaration
...
enum class Status { Invalid, Success, Fail };
Status doWork() // Full declaration required for implementation
{
    return Status::Success;
}
```

Unscoped enumerations:

```
...
enum Status: int; // Forward declaration, explicit type required
Status doWork(); // Use the forward declaration
...
enum Status: int{ Invalid=0, Success, Fail }; // Must match forward declare type
static_assert( Success == 1 );
```

An in-depth multi-file example can be found here: Blind fruit merchant example

Section 68.4: Basic Enumeration Declaration

Standard enumerations allow users to declare a useful name for a set of integers. The names are collectively referred to as enumerators. An enumeration and its associated enumerators are defined as follows:

```
enum myEnum
{
    enumName1,
    enumName2,
};
```

An enumeration is a *type*, one which is distinct from all other types. In this case, the name of this type is `myEnum`. Objects of this type are expected to assume the value of an enumerator within the enumeration.

The enumerators declared within the enumeration are constant values of the type of the enumeration. Though the enumerators are declared within the type, the scope operator `::` is not needed to access the name. So the name of the first enumerator is `enumName1`.

Version ≥ C++11

The scope operator can be optionally used to access an enumerator within an enumeration. So `enumName1` can also be spelled `myEnum::enumName1`.

Enumerators are assigned integer values starting from 0 and increasing by 1 for each enumerator in an enumeration. So in the above case, `enumName1` has the value 0, while `enumName2` has the value 1.

Enumerators can also be assigned a specific value by the user; this value must be an integral constant expression. Enumerators who's values are not explicitly provided will have their value set to the value of the previous enumerator + 1.

```
enum myEnum
```

```
{
    enumName1 = 1, // value will be 1
    enumName2 = 2, // value will be 2
    enumName3,     // value will be 3, previous value + 1
    enumName4 = 7, // value will be 7
    enumName5,     // value will be 8
    enumName6 = 5, // value will be 5, legal to go backwards
    enumName7 = 3, // value will be 3, legal to reuse numbers
    enumName8 = enumName4 + 2, // value will be 9, legal to take prior enums and adjust them
};
```

Section 68.5: Enumeration in switch statements

A common use for enumerators is for switch statements and so they commonly appear in state machines. In fact a useful feature of switch statements with enumerations is that if no default statement is included for the switch, and not all values of the enum have been utilized, the compiler will issue a warning.

```
enum State {
    start,
    middle,
    end
};

...

switch(myState) {
    case start:
       ...
    case middle:
       ...
} // warning: enumeration value 'end' not handled in switch [-Wswitch]
```

Chapter 69: Iteration

Section 69.1: break

Jumps out of the nearest enclosing loop or `switch` statement.

```
// print the numbers to a file, one per line
for (const int num : num_list) {
    errno = 0;
    fprintf(file, "%d\n", num);
    if (errno == ENOSPC) {
        fprintf(stderr, "no space left on device; output will be truncated\n");
        break;
    }
}
```

Section 69.2: continue

Jumps to the end of the smallest enclosing loop.

```
int sum = 0;
for (int i = 0; i < N; i++) {
    int x;
    std::cin >> x;
    if (x < 0) continue;
    sum += x;
    // equivalent to: if (x >= 0) sum += x;
}
```

Section 69.3: do

Introduces a do-while loop.

```
// Gets the next non-whitespace character from standard input
char read_char() {
    char c;
    do {
        c = getchar();
    } while (isspace(c));
    return c;
}
```

Section 69.4: while

Introduces a while loop.

```
int i = 0;
// print 10 asterisks
while (i < 10) {
    putchar('*');
    i++;
}
```

Section 69.5: range-based for loop

```
std::vector<int> primes = {2, 3, 5, 7, 11, 13};

for(auto prime : primes) {
    std::cout << prime << std::endl;
}
```

Section 69.6: for

Introduces a for loop or, in C++11 and later, a range-based for loop.

```
// print 10 asterisks
for (int i = 0; i < 10; i++) {
    putchar('*');
}
```

Chapter 70: Regular expressions

Signature	Description
`bool regex_match(BidirectionalIterator first, BidirectionalIterator last, smatch& sm, const regex& re, regex_constraints::match_flag_type flags)`	**`BidirectionalIterator`** is any character iterator that provides increment and decrement operators **`smatch`** may be `cmatch` or any other other variant of `match_results` that accepts the type of `BidirectionalIterator` the `smatch` argument may be ommitted if the results of the regex are not needed **Returns** whether `re` matches the entire character sequence defined by `first` and `last`
`bool regex_match(const string& str, smatch& sm, const regex re&, regex_constraints::match_flag_type flags)`	**`string`** may be either a `const char*` or an L-Value `string`, *the functions accepting an R-Value `string` are explicitly deleted* **`smatch`** may be `cmatch` or any other other variant of `match_results` that accepts the type of `str` the `smatch` argument may be ommitted if the results of the regex are not needed **Returns** whether `re` matches the entire character sequence defined by `str`

Regular Expressions (sometimes called regexs or regexps) are a textual syntax which represents the patterns which can be matched in the strings operated upon.

Regular Expressions, introduced in c++11, may optionally support a return array of matched strings or another textual syntax defining how to replace matched patterns in strings operated upon.

Section 70.1: Basic regex_match and regex_search Examples

```
const auto input = "Some people, when confronted with a problem, think \"I know, I'll use regular
expressions.\""s;
smatch sm;

cout << input << endl;

// If input ends in a quotation that contains a word that begins with "reg" and another word
beginning with "ex" then capture the preceding portion of input
if (regex_match(input, sm, regex("(.*)\".*\\breg.*\\bex.*\"\\s*$"))) {
    const auto capture = sm[1].str();

    cout << '\t' << capture << endl; // Outputs: "\tSome people, when confronted with a problem,
think\n"

    // Search our capture for "a problem" or "# problems"
    if(regex_search(capture, sm, regex("(a|d+)\\s+problems?"))) {
        const auto count = sm[1] == "a"s ? 1 : stoi(sm[1]);

        cout << '\t' << count << (count > 1 ? " problems\n" : " problem\n"); // Outputs: "\t1
problem\n"
        cout << "Now they have " << count + 1 << " problems.\n"; // Outputs: "Now they have 2
problems\n"
    }
}
```

Live Example

Section 70.2: regex_iterator Example

When processing of captures has to be done iteratively a `regex_iterator` is a good choice. Dereferencing a `regex_iterator` returns a `match_result`. This is great for conditional captures or captures which have

interdependence. Let's say that we want to tokenize some C++ code. Given:

```
enum TOKENS {
    NUMBER,
    ADDITION,
    SUBTRACTION,
    MULTIPLICATION,
    DIVISION,
    EQUALITY,
    OPEN_PARENTHESIS,
    CLOSE_PARENTHESIS
};
```

We can tokenize this string: `const auto input = "42/2 + -8\t=\n(2 + 2) * 2 * 2 -3"s` with a `regex_iterator` like this:

```
vector<TOKENS> tokens;
const regex re{ "\\s*(\\(?)\\s*(-?\\s*\\d+)\\s*(\\)?)\\s*(?:(\\+)|(-)|(\\*)|(/)|(=))" };

for_each(sregex_iterator(cbegin(input), cend(input), re), sregex_iterator(), [&](const auto& i) {
    if(i[1].length() > 0) {
        tokens.push_back(OPEN_PARENTHESIS);
    }

    tokens.push_back(i[2].str().front() == '-' ? NEGATIVE_NUMBER : NON_NEGATIVE_NUMBER);

    if(i[3].length() > 0) {
        tokens.push_back(CLOSE_PARENTHESIS);
    }

    auto it = next(cbegin(i), 4);

    for(int result = ADDITION; it != cend(i); ++result, ++it) {
        if (it->length() > 0U) {
            tokens.push_back(static_cast<TOKENS>(result));
            break;
        }
    }
});

match_results<string::const_reverse_iterator> sm;

if(regex_search(crbegin(input), crend(input), sm, regex{ tokens.back() == SUBTRACTION ?
"^\\s*\\d+\\s*-\\s*(-?)" : "^\\s*\\d+\\s*(-?)" })) {
    tokens.push_back(sm[1].length() == 0 ? NON_NEGATIVE_NUMBER : NEGATIVE_NUMBER);
}
```

Live Example

A notable gotcha with regex iterators is that the `regex` argument must be an L-value, an R-value will not work: Visual Studio regex_iterator Bug?

Section 70.3: Anchors

C++ provides only 4 anchors:

- `^` which asserts the start of the string
- `$` which asserts the end of the string
- `\b` which asserts a `\W` character or the beginning or end of the string

- `\B` which asserts a `\w` character

Let's say for example we want to capture a number *with* it's sign:

```
auto input = "+1--12*123/+1234"s;
smatch sm;

if(regex_search(input, sm, regex{ "(?:^|\\b\\W)([+-]?\\d+)" })) {

    do {
        cout << sm[1] << endl;
        input = sm.suffix().str();
    } while(regex_search(input, sm, regex{ "(?:^\\W|\\b\\W)([+-]?\\d+)" }));
}
```

Live Example

An important note here is that the anchor does not consume any characters.

Section 70.4: regex_replace Example

This code takes in various brace styles and converts them to One True Brace Style:

```
const auto input = "if (KnR)\n\tfoo();\nif (spaces) {\n    foo();\n}\nif
(allman)\n{\n\tfoo();\n}\nif (horstmann)\n{\tfoo();\n}\nif (pico)\n{\tfoo(); }\nif
(whitesmiths)\n\t{\n\tfoo();\n\t}\n"s;

cout << input << regex_replace(input, regex("(.+?)\\s*\\{?\\s*(.+?;)\\s*\\}?\\s*"), "$1
{\n\t$2\n}\n") << endl;
```

Live Example

Section 70.5: regex_token_iterator Example

A `std::regex_token_iterator` provides a tremendous tool for extracting elements of a Comma Separated Value file. Aside from the advantages of iteration, this iterator is also able to capture escaped commas where other methods struggle:

```
const auto input = "please split,this,csv, ,line,\\,\n"s;
const regex re{ "((?:[^\\\\,]|\\\\.)+)(?:,|$)" };
const vector<string> m_vecFields{ sregex_token_iterator(cbegin(input), cend(input), re, 1),
sregex_token_iterator() };

cout << input << endl;

copy(cbegin(m_vecFields), cend(m_vecFields), ostream_iterator<string>(cout, "\n"));
```

Live Example

A notable gotcha with regex iterators is, that the `regex` argument must be an L-value. An R-value will not work.

Section 70.6: Quantifiers

Let's say that we're given `const string input` as a phone number to be validated. We could start by requiring a numeric input with a **zero or more quantifier**: `regex_match(input, regex("\\d*"))` or a **one or more quantifier**: `regex_match(input, regex("\\d+"))` But both of those really fall short if `input` contains an invalid

numeric string like: "123" Let's use a **n or more quantifier** to ensure that we're getting at least 7 digits:

```
regex_match(input, regex("\\d{7,}"))
```

This will guarantee that we will get at least a phone number of digits, but `input` could also contain a numeric string that's too long like: "123456789012". So lets go with a **between n and m quantifier** so the `input` is at least 7 digits but not more than 11:

```
regex_match(input, regex("\\d{7,11}"));
```

This gets us closer, but illegal numeric strings that are in the range of [7, 11] are still accepted, like: "123456789" So let's make the country code optional with a **lazy quantifier**:

```
regex_match(input, regex("\\d?\\d{7,10}"))
```

It's important to note that the **lazy quantifier** matches *as few characters as possible*, so the only way this character will be matched is if there are already 10 characters that have been matched by `\d{7,10}`. (To match the first character greedily we would have had to do: `\d{0,1}`.) The **lazy quantifier** can be appended to any other quantifier.

Now, how would we make the area code optional *and* only accept a country code if the area code was present?

```
regex_match(input, regex("(?:\\d{3,4})?\\d{7}"))
```

In this final regex, the `\d{7}` *requires* 7 digits. These 7 digits are optionally preceded by either 3 or 4 digits.

Note that we did not append the **lazy quantifier**: `\d{3,4}?\d{7}`, the `\d{3,4}?` would have matched either 3 or 4 characters, preferring 3. Instead we're making the non-capturing group match at most once, preferring not to match. Causing a mismatch if `input` didn't include the area code like: "1234567".

In conclusion of the quantifier topic, I'd like to mention the other appending quantifier that you can use, the **possessive quantifier**. *Either* the **lazy quantifier** or the **possessive quantifier** can be appended to any quantifier. The **possessive quantifier**'s only function is to assist the regex engine by telling it, greedily take these characters *and don't ever give them up even if it causes the regex to fail*. This for example doesn't make much sense: `regex_match(input, regex("\\d{3,4}+\\d{7}))` Because an `input` like: "1234567890" wouldn't be matched as `\d{3,4}+` will always match 4 characters even if matching 3 would have allowed the regex to succeed.
The **possessive quantifier** is best used *when the quantified token limits the number of matchable characters*. For example:

```
regex_match(input, regex("(?:.*\\d{3,4}+){3}"))
```

Can be used to match if `input` contained any of the following:

> 123 456 7890
> 123-456-7890
> (123)456-7890
> (123) 456 - 7890

But when this regex really shines is when `input` contains an *illegal* input:

> 12345 - 67890

Without the **possessive quantifier** the regex engine has to go back and test *every combination of* `.*` *and either 3 or 4 characters* to see if it can find a matchable combination. With the **possessive quantifier** the regex starts where the 2nd **possessive quantifier** left off, the '0' character, and the regex engine tries to adjust the `.*` to allow `\d{3,4}` to match; when it can't the regex just fails, no back tracking is done to see if earlier `.*` adjustment could have allowed a match.

Section 70.7: Splitting a string

```
std::vector<std::string> split(const std::string &str, std::string regex)
{
    std::regex r{ regex };
    std::sregex_token_iterator start{ str.begin(), str.end(), r, -1 }, end;
    return std::vector<std::string>(start, end);
}
```

```
split("Some  string\t with whitespace ", "\\s+"); // "Some", "string", "with", "whitespace"
```

Chapter 71: Implementation-defined behavior

Section 71.1: Size of integral types

The following types are defined as *integral types*:

- `char`
- Signed integer types
- Unsigned integer types
- `char16_t` and `char32_t`
- `bool`
- `wchar_t`

With the exception of `sizeof(char)` / `sizeof(signed char)` / `sizeof(unsigned char)`, which is split between § 3.9.1.1 [basic.fundamental/1] and § 5.3.3.1 [expr.sizeof], and `sizeof(bool)`, which is entirely implementation-defined and has no minimum size, the minimum size requirements of these types are given in section § 3.9.1 [basic.fundamental] of the standard, and shall be detailed below.

Size of `char`

All versions of the C++ standard specify, in § 5.3.3.1, that `sizeof` yields 1 for `unsigned char`, `signed char`, and `char` (it is implementation defined whether the `char` type is `signed` or `unsigned`).

Version ≥ C++14

`char` is large enough to represent 256 different values, to be suitable for storing UTF-8 code units.

Size of signed and unsigned integer types

The standard specifies, in § 3.9.1.2, that in the list of *standard signed integer types*, consisting of `signed char`, `short int`, `int`, `long int`, and `long long int`, each type will provide at least as much storage as those preceding it in the list. Furthermore, as specified in § 3.9.1.3, each of these types has a corresponding *standard unsigned integer type*, `unsigned char`, `unsigned short int`, `unsigned int`, `unsigned long int`, and `unsigned long long int`, which has the same size and alignment as its corresponding signed type. Additionally, as specified in § 3.9.1.1, `char` has the same size and alignment requirements as both `signed char` and `unsigned char`.

Version < C++11

Prior to C++11, `long long` and `unsigned long long` were not officially part of the C++ standard. However, after their introduction to C, in C99, many compilers supported `long long` as an *extended signed integer type*, and `unsigned long long` as an *extended unsigned integer type*, with the same rules as the C types.

The standard thus guarantees that:

```
1 == sizeof(char)  == sizeof(signed char) == sizeof(unsigned char)
  <= sizeof(short) == sizeof(unsigned short)
  <= sizeof(int)   == sizeof(unsigned int)
  <= sizeof(long)  == sizeof(unsigned long)
```

Version ≥ C++11

```
 <= sizeof(long long) == sizeof(unsigned long long)
```

Specific minimum sizes for each type are not given by the standard. Instead, each type has a minimum range of

values it can support, which is, as specified in § 3.9.1.3, inherited from the C standard, in §5.2.4.2.1. The minimum size of each type can be roughly inferred from this range, by determining the minimum number of bits required; note that for any given platform, any type's actual supported range may be larger than the minimum. Note that for signed types, ranges correspond to one's complement, not the more commonly used two's complement; this is to allow a wider range of platforms to comply with the standard.

Type	Minimum range	Minimum bits required
`signed char`	-127 to 127 (-(27 - 1) to (27 - 1))	8
`unsigned char`	0 to 255 (0 to 28 - 1)	8
`signed short`	-32,767 to 32,767 (-(215 - 1) to (215 - 1))	16
`unsigned short`	0 to 65,535 (0 to 216 - 1)	16
`signed int`	-32,767 to 32,767 (-(215 - 1) to (215 - 1))	16
`unsigned int`	0 to 65,535 (0 to 216 - 1)	16
`signed long`	-2,147,483,647 to 2,147,483,647 (-(231 - 1) to (231 - 1))	32
`unsigned long`	0 to 4,294,967,295 (0 to 232 - 1)	32

Version ≥ C++11

Type	Minimum range	Minimum bits required
`signed long long`	-9,223,372,036,854,775,807 to 9,223,372,036,854,775,807 (-(263 - 1) to (263 - 1))	64
`unsigned long long`	0 to 18,446,744,073,709,551,615 (0 to 264 - 1)	64

As each type is allowed to be greater than its minimum size requirement, types may differ in size between implementations. The most notable example of this is with the 64-bit data models LP64 and LLP64, where LLP64 systems (such as 64-bit Windows) have 32-bit `int`s and `long`s, and LP64 systems (such as 64-bit Linux) have 32-bit `int`s and 64-bit `long`s. Due to this, integer types cannot be assumed to have a fixed width across all platforms.

Version ≥ C++11

If integer types with fixed width are required, use types from the `<cstdint>` header, but note that the standard makes it optional for implementations to support the exact-width types `int8_t`, `int16_t`, `int32_t`, `int64_t`, `intptr_t`, `uint8_t`, `uint16_t`, `uint32_t`, `uint64_t` and `uintptr_t`.

Version ≥ C++11

Size of `char16_t` and `char32_t`

The sizes of `char16_t` and `char32_t` are implementation-defined, as specified in § 5.3.3.1, with the stipulations given in § 3.9.1.5:

- `char16_t` is large enough to represent any UTF-16 code unit, and has the same size, signedness, and alignment as `uint_least16_t`; it is thus required to be at least 16 bits in size.
- `char32_t` is large enough to represent any UTF-32 code unit, and has the same size, signedness, and alignment as `uint_least32_t`; it is thus required to be at least 32 bits in size.

Size of `bool`

The size of `bool` is implementation defined, and may or may not be 1.

Size of wchar_t

`wchar_t`, as specified in § 3.9.1.5, is a distinct type, whose range of values can represent every distinct code unit of the largest extended character set among the supported locales. It has the same size, signedness, and alignment as one of the other integral types, which is known as its *underlying type*. This type's size is implementation-defined, as

specified in § 5.3.3.1, and may be, for example, at least 8, 16, or 32 bits; if a system supports Unicode, for example, `wchar_t` is required to be at least 32 bits (an exception to this rule is Windows, where `wchar_t` is 16 bits for compatibility purposes). It is inherited from the C90 standard, ISO 9899:1990 § 4.1.5, with only minor rewording.

Depending on the implementation, the size of `wchar_t` is often, but not always, 8, 16, or 32 bits. The most common examples of these are:

- In Unix and Unix-like systems, `wchar_t` is 32-bit, and is usually used for UTF-32.
- In Windows, `wchar_t` is 16-bit, and is used for UTF-16.
- On a system which only has 8-bit support, `wchar_t` is 8 bit.

Version ≥ C++11

If Unicode support is desired, it is recommended to use `char` for UTF-8, `char16_t` for UTF-16, or `char32_t` for UTF-32, instead of using `wchar_t`.

Data Models

As mentioned above, the widths of integer types can differ between platforms. The most common models are as follows, with sizes specified in bits:

Model	`int`	`long`	pointer
LP32 (2/4/4)	16	32	32
ILP32 (4/4/4)	32	32	32
LLP64 (4/4/8)	32	32	64
LP64 (4/8/8)	32	64	64

Out of these models:

- 16-bit Windows used LP32.
- 32-bit *nix systems (Unix, Linux, Mac OSX, and other Unix-like OSes) and Windows use ILP32.
- 64-bit Windows uses LLP64.
- 64-bit *nix systems use LP64.

Note, however, that these models aren't specifically mentioned in the standard itself.

Section 71.2: Char might be unsigned or signed

The standard doesn't specify if `char` should be signed or unsigned. Different compilers implement it differently, or might allow to change it using a command line switch.

Section 71.3: Ranges of numeric types

The ranges of the integer types are implementation-defined. The header `<limits>` provides the `std::numeric_limits<T>` template which provides the minimum and maximum values of all fundamental types. The values satisfy guarantees provided by the C standard through the `<climits>` and (>= C++11) `<cinttypes>` headers.

- `std::numeric_limits<signed char>::min()` equals `SCHAR_MIN`, which is less than or equal to -127.
- `std::numeric_limits<signed char>::max()` equals `SCHAR_MAX`, which is greater than or equal to 127.
- `std::numeric_limits<unsigned char>::max()` equals `UCHAR_MAX`, which is greater than or equal to 255.
- `std::numeric_limits<short>::min()` equals `SHRT_MIN`, which is less than or equal to -32767.
- `std::numeric_limits<short>::max()` equals `SHRT_MAX`, which is greater than or equal to 32767.

- `std::numeric_limits<unsigned short>::max()` equals `USHRT_MAX`, which is greater than or equal to 65535.
- `std::numeric_limits<int>::min()` equals `INT_MIN`, which is less than or equal to -32767.
- `std::numeric_limits<int>::max()` equals `INT_MAX`, which is greater than or equal to 32767.
- `std::numeric_limits<unsigned int>::max()` equals `UINT_MAX`, which is greater than or equal to 65535.
- `std::numeric_limits<long>::min()` equals `LONG_MIN`, which is less than or equal to -2147483647.
- `std::numeric_limits<long>::max()` equals `LONG_MAX`, which is greater than or equal to 2147483647.
- `std::numeric_limits<unsigned long>::max()` equals `ULONG_MAX`, which is greater than or equal to 4294967295.

Version ≥ C++11

- `std::numeric_limits<long long>::min()` equals `LLONG_MIN`, which is less than or equal to -9223372036854775807.
- `std::numeric_limits<long long>::max()` equals `LLONG_MAX`, which is greater than or equal to 9223372036854775807.
- `std::numeric_limits<unsigned long long>::max()` equals `ULLONG_MAX`, which is greater than or equal to 18446744073709551615.

For floating-point types `T`, `max()` is the maximum finite value while `min()` is the minimum positive normalized value. Additional members are provided for floating-point types, which are also implementation-defined but satisfy certain guarantees provided by the C standard through the **`<cfloat>`** header.

- The member `digits10` gives the number of decimal digits of precision.
  - `std::numeric_limits<float>::digits10` equals `FLT_DIG`, which is at least 6.
  - `std::numeric_limits<double>::digits10` equals `DBL_DIG`, which is at least 10.
  - `std::numeric_limits<long double>::digits10` equals `LDBL_DIG`, which is at least 10.
- The member `min_exponent10` is the minimum negative E such that 10 to the power E is normal.
  - `std::numeric_limits<float>::min_exponent10` equals `FLT_MIN_10_EXP`, which is at most -37.
  - `std::numeric_limits<double>::min_exponent10` equals `DBL_MIN_10_EXP`, which is at most -37.
    `std::numeric_limits<long double>::min_exponent10` equals `LDBL_MIN_10_EXP`, which is at most -37.
- The member `max_exponent10` is the maximum E such that 10 to the power E is finite.
  - `std::numeric_limits<float>::max_exponent10` equals `FLT_MIN_10_EXP`, which is at least 37.
  - `std::numeric_limits<double>::max_exponent10` equals `DBL_MIN_10_EXP`, which is at least 37.
  - `std::numeric_limits<long double>::max_exponent10` equals `LDBL_MIN_10_EXP`, which is at least 37.
- If the member `is_iec559` is true, the type conforms to IEC 559 / IEEE 754, and its range is therefore determined by that standard.

Section 71.4: Value representation of floating point types

The standard requires that `long double` provides at least as much precision as `double`, which provides at least as much precision as `float`; and that a `long double` can represent any value that a `double` can represent, while a `double` can represent any value that a `float` can represent. The details of the representation are, however, implementation-defined.

For a floating point type `T`, `std::numeric_limits<T>::radix` specifies the radix used by the representation of `T`.

If `std::numeric_limits<T>::is_iec559` is true, then the representation of `T` matches one of the formats defined by IEC 559 / IEEE 754.

Section 71.5: Overflow when converting from integer to signed integer

When either a signed or unsigned integer is converted to a signed integer type, and its value is not representable in

the destination type, the value produced is implementation-defined. Example:

```
// Suppose that on this implementation, the range of signed char is -128 to +127 and
// the range of unsigned char is 0 to 255
int x = 12345;
signed char sc = x;   // sc has an implementation-defined value
unsigned char uc = x; // uc is initialized to 57 (i.e., 12345 modulo 256)
```

Section 71.6: Underlying type (and hence size) of an enum

If the underlying type is not explicitly specified for an unscoped enumeration type, it is determined in an implementation-defined manner.

```
enum E {
    RED,
    GREEN,
    BLUE,
};
using T = std::underlying_type<E>::type; // implementation-defined
```

However, the standard does require the underlying type of an enumeration to be no larger than `int` unless both `int` and `unsigned int` are unable to represent all the values of the enumeration. Therefore, in the above code, `T` could be `int`, `unsigned int`, or `short`, but not `long long`, to give a few examples.

Note that an enum has the same size (as returned by `sizeof`) as its underlying type.

Section 71.7: Numeric value of a pointer

The result of casting a pointer to an integer using `reinterpret_cast` is implementation-defined, but "... is intended to be unsurprising to those who know the addressing structure of the underlying machine."

```
int x = 42;
int* p = &x;
long addr = reinterpret_cast<long>(p);
std::cout << addr << "\n"; // prints some numeric address,
                           // probably in the architecture's native address format
```

Likewise, the pointer obtained by conversion from an integer is also implementation-defined.

The right way to store a pointer as an integer is using the `uintptr_t` or `intptr_t` types:

```
// `uintptr_t` was not in C++03. It's in C99, in <stdint.h>, as an optional type
#include <stdint.h>

uintptr_t uip;
```

Version ≥ C++11

```
// There is an optional `std::uintptr_t` in C++11
#include <cstdint>

std::uintptr_t uip;
```

C++11 refers to C99 for the definition `uintptr_t` (C99 standard, 6.3.2.3):

> an unsigned integer type with the property that any valid pointer to `void` can be converted to this type, then converted back to pointer to `void`, and the result will compare equal to the original pointer.

While, for the majority of modern platforms, you can assume a flat address space and that arithmetic on `uintptr_t` is equivalent to arithmetic on `char *`, it's entirely possible for an implementation to perform any transformation when casting `void *` to `uintptr_t` as long the transformation can be reversed when casting back from `uintptr_t` to `void *`.

Technicalities

- On XSI-conformant (X/Open System Interfaces) systems, `intptr_t` and `uintptr_t` types are required, otherwise they are **optional**.
- Within the meaning of the C standard, functions aren't objects; it isn't guaranteed by the C standard that `uintptr_t` can hold a function pointer. Anyway POSIX (2.12.3) conformance requires that:

  > All function pointer types shall have the same representation as the type pointer to void. Conversion of a function pointer to void * shall not alter the representation. A void * value resulting from such a conversion can be converted back to the original function pointer type, using an explicit cast, without loss of information.

- C99 §7.18.1:

  > When typedef names differing only in the absence or presence of the initial u are defined, they shall denote corresponding signed and unsigned types as described in 6.2.5; an implementation providing one of these corresponding types shall also provide the other.

  `uintptr_t` might make sense if you want to do things to the bits of the pointer that you can't do as sensibly with a signed integer.

Section 71.8: Number of bits in a byte

In C++, a *byte* is the space occupied by a `char` object. The number of bits in a byte is given by `CHAR_BIT`, which is defined in `climits` and required to be at least 8. While most modern systems have 8-bit bytes, and POSIX requires `CHAR_BIT` to be exactly 8, there are some systems where `CHAR_BIT` is greater than 8 i.e a single byte may be comprised of 8, 16, 32 or 64 bits.

Chapter 72: Exceptions

Section 72.1: Catching exceptions

A `try/catch` block is used to catch exceptions. The code in the `try` section is the code that may throw an exception, and the code in the `catch` clause(s) handles the exception.

```
#include <iostream>
#include <string>
#include <stdexcept>

int main() {
  std::string str("foo");

  try {
      str.at(10); // access element, may throw std::out_of_range
  } catch (const std::out_of_range& e) {
      // what() is inherited from std::exception and contains an explanatory message
      std::cout << e.what();
  }
}
```

Multiple `catch` clauses may be used to handle multiple exception types. If multiple `catch` clauses are present, the exception handling mechanism tries to match them **in order** of their appearance in the code:

```
std::string str("foo");

try {
    str.reserve(2); // reserve extra capacity, may throw std::length_error
    str.at(10); // access element, may throw std::out_of_range
} catch (const std::length_error& e) {
    std::cout << e.what();
} catch (const std::out_of_range& e) {
    std::cout << e.what();
}
```

Exception classes which are derived from a common base class can be caught with a single `catch` clause for the common base class. The above example can replace the two `catch` clauses for `std::length_error` and `std::out_of_range` with a single clause for `std:exception`:

```
std::string str("foo");

try {
    str.reserve(2); // reserve extra capacity, may throw std::length_error
    str.at(10); // access element, may throw std::out_of_range
} catch (const std::exception& e) {
    std::cout << e.what();
}
```

Because the `catch` clauses are tried in order, be sure to write more specific catch clauses first, otherwise your exception handling code might never get called:

```
try {
    /* Code throwing exceptions omitted. */
} catch (const std::exception& e) {
    /* Handle all exceptions of type std::exception. */
} catch (const std::runtime_error& e) {
```

```
    /* This block of code will never execute, because std::runtime_error inherits
       from std::exception, and all exceptions of type std::exception were already
       caught by the previous catch clause. */
}
```

Another possibility is the catch-all handler, which will catch any thrown object:

```
try {
    throw 10;
} catch (...) {
    std::cout << "caught an exception";
}
```

Section 72.2: Rethrow (propagate) exception

Sometimes you want to do something with the exception you catch (like write to log or print a warning) and let it bubble up to the upper scope to be handled. To do so, you can rethrow any exception you catch:

```
try {
    ... // some code here
} catch (const SomeException& e) {
    std::cout << "caught an exception";
    throw;
}
```

Using `throw;` without arguments will re-throw the currently caught exception.

Version ≥ C++11

To rethrow a managed `std::exception_ptr`, the C++ Standard Library has the `rethrow_exception` function that can be used by including the **`<exception>`** header in your program.

```
#include <iostream>
#include <string>
#include <exception>
#include <stdexcept>

void handle_eptr(std::exception_ptr eptr) // passing by value is ok
{
    try {
        if (eptr) {
            std::rethrow_exception(eptr);
        }
    } catch(const std::exception& e) {
        std::cout << "Caught exception \"" << e.what() << "\"\n";
    }
}

int main()
{
    std::exception_ptr eptr;
    try {
        std::string().at(1); // this generates an std::out_of_range
    } catch(...) {
        eptr = std::current_exception(); // capture
    }
    handle_eptr(eptr);
} // destructor for std::out_of_range called here, when the eptr is destructed
```

Section 72.3: Best practice: throw by value, catch by const reference

In general, it is considered good practice to throw by value (rather than by pointer), but catch by (const) reference.

```
try {
    // throw new std::runtime_error("Error!");   // Don't do this!
    // This creates an exception object
    // on the heap and would require you to catch the
    // pointer and manage the memory yourself. This can
    // cause memory leaks!

    throw std::runtime_error("Error!");
} catch (const std::runtime_error& e) {
    std::cout << e.what() << std::endl;
}
```

One reason why catching by reference is a good practice is that it eliminates the need to reconstruct the object when being passed to the catch block (or when propagating through to other catch blocks). Catching by reference also allows the exceptions to be handled polymorphically and avoids object slicing. However, if you are rethrowing an exception (like `throw e;`, see example below), you can still get object slicing because the `throw e;` statement makes a copy of the exception as whatever type is declared:

```
#include <iostream>

struct BaseException {
    virtual const char* what() const { return "BaseException"; }
};

struct DerivedException : BaseException {
    // "virtual" keyword is optional here
    virtual const char* what() const { return "DerivedException"; }
};

int main(int argc, char** argv) {
    try {
        try {
            throw DerivedException();
        } catch (const BaseException& e) {
            std::cout << "First catch block: " << e.what() << std::endl;
            // Output ==> First catch block: DerivedException

            throw e; // This changes the exception to BaseException
                     // instead of the original DerivedException!
        }
    } catch (const BaseException& e) {
        std::cout << "Second catch block: " << e.what() << std::endl;
        // Output ==> Second catch block: BaseException
    }
    return 0;
}
```

If you are sure that you are not going to do anything to change the exception (like add information or modify the message), catching by const reference allows the compiler to make optimizations and can improve performance. But this can still cause object splicing (as seen in the example above).

Warning: Beware of throwing unintended exceptions in `catch` blocks, especially related to allocating extra memory or resources. For example, constructing `logic_error`, `runtime_error` or their subclasses might throw `bad_alloc`

due to memory running out when copying the exception string, I/O streams might throw during logging with respective exception masks set, etc.

Section 72.4: Custom exception

You shouldn't throw raw values as exceptions, instead use one of the standard exception classes or make your own.

Having your own exception class inherited from `std::exception` is a good way to go about it. Here's a custom exception class which directly inherits from `std::exception`:

```
#include <exception>

class Except: virtual public std::exception {

protected:

    int error_number;               ///< Error number
    int error_offset;               ///< Error offset
    std::string error_message;      ///< Error message

public:

    /** Constructor (C++ STL string, int, int).
     *  @param msg The error message
     *  @param err_num Error number
     *  @param err_off Error offset
     */
    explicit
    Except(const std::string& msg, int err_num, int err_off):
        error_number(err_num),
        error_offset(err_off),
        error_message(msg)
        {}

    /** Destructor.
     *  Virtual to allow for subclassing.
     */
    virtual ~Except() throw () {}

    /** Returns a pointer to the (constant) error description.
     *  @return A pointer to a const char*. The underlying memory
     *  is in possession of the Except object. Callers must
     *  not attempt to free the memory.
     */
    virtual const char* what() const throw () {
       return error_message.c_str();
    }

    /** Returns error number.
     *  @return #error_number
     */
    virtual int getErrorNumber() const throw() {
        return error_number;
    }

    /**Returns error offset.
     * @return #error_offset
     */
    virtual int getErrorOffset() const throw() {
```

```
        return error_offset;
    }

};
```

An example throw catch:

```
try {
    throw(Except("Couldn't do what you were expecting", -12, -34));
} catch (const Except& e) {
    std::cout<<e.what()
             <<"\nError number: "<<e.getErrorNumber()
             <<"\nError offset: "<<e.getErrorOffset();
}
```

As you are not only just throwing a dumb error message, also some other values representing what the error exactly was, your error handling becomes much more efficient and meaningful.

There's an exception class that let's you handle error messages nicely :`std::runtime_error`

You can inherit from this class too:

```
#include <stdexcept>

class Except: virtual public std::runtime_error {

protected:

    int error_number;               ///< Error number
    int error_offset;               ///< Error offset

public:

    /** Constructor (C++ STL string, int, int).
     *  @param msg The error message
     *  @param err_num Error number
     *  @param err_off Error offset
     */
    explicit
    Except(const std::string& msg, int err_num, int err_off):
        std::runtime_error(msg)
        {
            error_number = err_num;
            error_offset = err_off;

        }

    /** Destructor.
     *  Virtual to allow for subclassing.
     */
    virtual ~Except() throw () {}

    /** Returns error number.
     *  @return #error_number
     */
    virtual int getErrorNumber() const throw() {
        return error_number;
    }

    /**Returns error offset.
```

```
     * @return #error_offset
     */
    virtual int getErrorOffset() const throw() {
        return error_offset;
    }

};
```

Note that I haven't overridden the `what()` function from the base class (`std::runtime_error`) i.e we will be using the base class's version of `what()`. You can override it if you have further agenda.

Section 72.5: std::uncaught_exceptions

Version ≥ c++17

C++17 introduces `int std::uncaught_exceptions()` (to replace the limited `bool std::uncaught_exception()`) to know how many exceptions are currently uncaught. That allows for a class to determine if it is destroyed during a stack unwinding or not.

```
#include <exception>
#include <string>
#include <iostream>

// Apply change on destruction:
// Rollback in case of exception (failure)
// Else Commit (success)
class Transaction
{
public:
    Transaction(const std::string& s) : message(s) {}
    Transaction(const Transaction&) = delete;
    Transaction& operator =(const Transaction&) = delete;
    void Commit() { std::cout << message << ": Commit\n"; }
    void RollBack() noexcept(true) { std::cout << message << ": Rollback\n"; }

    // ...

    ~Transaction() {
        if (uncaughtExceptionCount == std::uncaught_exceptions()) {
            Commit(); // May throw.
        } else { // current stack unwinding
            RollBack();
        }
    }

private:
    std::string message;
    int uncaughtExceptionCount = std::uncaught_exceptions();
};

class Foo
{
public:
    ~Foo() {
        try {
            Transaction transaction("In ~Foo"); // Commit,
                                            // even if there is an uncaught exception
            //...
        } catch (const std::exception& e) {
            std::cerr << "exception/~Foo:" << e.what() << std::endl;
```

```
        }
    }
};

int main()
{
    try {
        Transaction transaction("In main"); // RollBack
        Foo foo; // ~Foo commit its transaction.
        //...
        throw std::runtime_error("Error");
    } catch (const std::exception& e) {
        std::cerr << "exception/main:" << e.what() << std::endl;
    }
}
```

Output:

```
In ~Foo: Commit
In main: Rollback
exception/main:Error
```

Section 72.6: Function Try Block for regular function

```
void function_with_try_block()
try
{
    // try block body
}
catch (...)
{
    // catch block body
}
```

Which is equivalent to

```
void function_with_try_block()
{
    try
    {
        // try block body
    }
    catch (...)
    {
        // catch block body
    }
}
```

Note that for constructors and destructors, the behavior is different as the catch block re-throws an exception anyway (the caught one if there is no other throw in the catch block body).

The function `main` is allowed to have a function try block like any other function, but `main`'s function try block will not catch exceptions that occur during the construction of a non-local static variable or the destruction of any static variable. Instead, `std::terminate` is called.

Section 72.7: Nested exception

Version ≥ C++11

During exception handling there is a common use case when you catch a generic exception from a low-level function (such as a filesystem error or data transfer error) and throw a more specific high-level exception which indicates that some high-level operation could not be performed (such as being unable to publish a photo on Web). This allows exception handling to react to specific problems with high level operations and also allows, having only error an message, the programmer to find a place in the application where an exception occurred. Downside of this solution is that exception callstack is truncated and original exception is lost. This forces developers to manually include text of original exception into a newly created one.

Nested exceptions aim to solve the problem by attaching low-level exception, which describes the cause, to a high level exception, which describes what it means in this particular case.

`std::nested_exception` allows to nest exceptions thanks to `std::throw_with_nested`:

```
#include <stdexcept>
#include <exception>
#include <string>
#include <fstream>
#include <iostream>

struct MyException
{
    MyException(const std::string& message) : message(message) {}
    std::string message;
};

void print_current_exception(int level)
{
    try {
        throw;
    } catch (const std::exception& e) {
        std::cerr << std::string(level, ' ') << "exception: " << e.what() << '\n';
    } catch (const MyException& e) {
        std::cerr << std::string(level, ' ') << "MyException: " << e.message << '\n';
    } catch (...) {
        std::cerr << "Unkown exception\n";
    }
}

void print_current_exception_with_nested(int level =  0)
{
    try {
        throw;
    } catch (...) {
        print_current_exception(level);
    }
    try {
        throw;
    } catch (const std::nested_exception& nested) {
        try {
            nested.rethrow_nested();
        } catch (...) {
            print_current_exception_with_nested(level + 1); // recursion
        }
    } catch (...) {
        //Empty // End recursion
    }
}

// sample function that catches an exception and wraps it in a nested exception
void open_file(const std::string& s)
```

```
{
    try {
        std::ifstream file(s);
        file.exceptions(std::ios_base::failbit);
    } catch(...) {
        std::throw_with_nested(MyException{"Couldn't open " + s});
    }
}

// sample function that catches an exception and wraps it in a nested exception
void run()
{
    try {
        open_file("nonexistent.file");
    } catch(...) {
        std::throw_with_nested( std::runtime_error("run() failed") );
    }
}

// runs the sample function above and prints the caught exception
int main()
{
    try {
        run();
    } catch(...) {
        print_current_exception_with_nested();
    }
}
```

Possible output:

```
exception: run() failed
 MyException: Couldn't open nonexistent.file
  exception: basic_ios::clear
```

If you work only with exceptions inherited from `std::exception`, code can even be simplified.

Section 72.8: Function Try Blocks In constructor

The only way to catch exception in initializer list:

```
struct A : public B
{
    A() try : B(), foo(1), bar(2)
    {
        // constructor body
    }
    catch (...)
    {
        // exceptions from the initializer list and constructor are caught here
        // if no exception is thrown here
        // then the caught exception is re-thrown.
    }

private:
    Foo foo;
    Bar bar;
};
```

Section 72.9: Function Try Blocks In destructor

```
struct A
{
    ~A() noexcept(false) try
    {
        // destructor body
    }
    catch (...)
    {
        // exceptions of destructor body are caught here
        // if no exception is thrown here
        // then the caught exception is re-thrown.
    }
};
```

Note that, although this is possible, one needs to be very careful with throwing from destructor, as if a destructor called during stack unwinding throws an exception, `std::terminate` is called.

Chapter 73: Lambdas

Parameter	Details
default-capture	Specifies how all non-listed variables are captured. Can be `=` (capture by value) or `&` (capture by reference). If omitted, non-listed variables are inaccessible within the *lambda-body*. The *default-capture* must precede the *capture-list*.
capture-list	Specifies how local variables are made accessible within the *lambda-body*. Variables without prefix are captured by value. Variables prefixed with `&` are captured by reference. Within a class method, `this` can be used to make all its members accessible by reference. Non-listed variables are inaccessible, unless the list is preceded by a *default-capture*.
argument-list	Specifies the arguments of the lambda function.
mutable	*(optional)* Normally variables captured by value are `const`. Specifying `mutable` makes them non-const. Changes to those variables are retained between calls.
throw-specification	*(optional)* Specifies the exception throwing behavior of the lambda function. For example: `noexcept` or `throw(std::exception)`.
attributes	*(optional)* Any attributes for the lambda function. For example, if the *lambda-body* always throws an exception then `[[noreturn]]` can be used.
-> return-type	*(optional)* Specifies the return type of the lambda function. Required when the return type cannot be determined by the compiler.
lambda-body	A code block containing the implementation of the lambda function.

Section 73.1: What is a lambda expression?

A **lambda expression** provides a concise way to create simple function objects. A lambda expression is a prvalue whose result object is called closure object, which behaves like a function object.

The name 'lambda expression' originates from lambda calculus, which is a mathematical formalism invented in the 1930s by Alonzo Church to investigate questions about logic and computability. Lambda calculus formed the basis of LISP, a functional programming language. Compared to lambda calculus and LISP, C++ lambda expressions share the properties of being unnamed, and to capture variables from the surrounding context, but they lack the ability to operate on and return functions.

A lambda expression is often used as an argument to functions that take a callable object. That can be simpler than creating a named function, which would be only used when passed as the argument. In such cases, lambda expressions are generally preferred because they allow defining the function objects inline.

A lambda consists typically of three parts: a capture list `[]`, an optional parameter list `()` and a body `{}`, all of which can be empty:

```
[](){}                // An empty lambda, which does and returns nothing
```

Capture list

`[]` is the **capture list**. By default, variables of the enclosing scope cannot be accessed by a lambda. *Capturing* a variable makes it accessible inside the lambda, either as a copy or as a reference. Captured variables become a part of the lambda; in contrast to function arguments, they do not have to be passed when calling the lambda.

```
int a = 0;                       // Define an integer variable
auto f = []()   { return a*9; }; // Error: 'a' cannot be accessed
auto f = [a]()  { return a*9; }; // OK, 'a' is "captured" by value
auto f = [&a]() { return a++; }; // OK, 'a' is "captured" by reference
                                 //      Note: It is the responsibility of the programmer
                                 //      to ensure that a is not destroyed before the
```

```
                                    //      lambda is called.
auto b = f();                       // Call the lambda function. a is taken from the capture list and
not passed here.
```

Parameter list

`()` is the **parameter list**, which is almost the same as in regular functions. If the lambda takes no arguments, these parentheses can be omitted (except if you need to declare the lambda `mutable`). These two lambdas are equivalent:

```
auto call_foo  = [x](){ x.foo(); };
auto call_foo2 = [x]{ x.foo(); };
```

Version ≥ C++14

The parameter list can use the placeholder type `auto` instead of actual types. By doing so, this argument behaves like a template parameter of a function template. Following lambdas are equivalent when you want to sort a vector in generic code:

```
auto sort_cpp11 = [](std::vector<T>::const_reference lhs, std::vector<T>::const_reference rhs) {
return lhs < rhs; };
auto sort_cpp14 = [](const auto &lhs, const auto &rhs) { return lhs < rhs; };
```

Function body

`{}` is the **body**, which is the same as in regular functions.

Calling a lambda

A lambda expression's result object is a closure, which can be called using the `operator()` (as with other function objects):

```
int multiplier = 5;
auto timesFive = [multiplier](int a) { return a * multiplier; };
std::out << timesFive(2); // Prints 10

multiplier = 15;
std::out << timesFive(2); // Still prints 2*5 == 10
```

Return Type

By default, the return type of a lambda expression is deduced.

```
[](){ return true; };
```

In this case the return type is `bool`.

You can also manually specify the return type using the following syntax:

```
[]() -> bool { return true; };
```

Mutable Lambda

Objects captured by value in the lambda are by default immutable. This is because the `operator()` of the generated closure object is `const` by default.

```
auto func = [c = 0](){++c; std::cout << c;};  // fails to compile because ++c
                                              // tries to mutate the state of
```

```
                                         // the lambda.
```

Modifying can be allowed by using the keyword `mutable`, which make the closer object's `operator()` non-`const`:

```
auto func = [c = 0]() mutable {++c; std::cout << c;};
```

If used together with the return type, `mutable` comes before it.

```
auto func = [c = 0]() mutable -> int {++c; std::cout << c; return c;};
```

An example to illustrate the usefulness of lambdas

Before C++11:

Version < C++11

```
// Generic functor used for comparison
struct islessthan
{
    islessthan(int threshold) : _threshold(threshold) {}

    bool operator()(int value) const
    {
        return value < _threshold;
    }
private:
    int _threshold;
};

// Declare a vector
const int arr[] = { 1, 2, 3, 4, 5 };
std::vector<int> vec(arr, arr+5);

// Find a number that's less than a given input (assume this would have been function input)
int threshold = 10;
std::vector<int>::iterator it = std::find_if(vec.begin(), vec.end(), islessthan(threshold));
```

Since C++11:

Version ≥ C++11

```
// Declare a vector
std::vector<int> vec{ 1, 2, 3, 4, 5 };

// Find a number that's less than a given input (assume this would have been function input)
int threshold = 10;
auto it = std::find_if(vec.begin(), vec.end(), [threshold](int value) { return value < threshold;
});
```

Section 73.2: Specifying the return type

For lambdas with a single return statement, or multiple return statements whose expressions are of the same type, the compiler can deduce the return type:

```
// Returns bool, because "value > 10" is a comparison which yields a Boolean result
auto l = [](int value) {
    return value > 10;
}
```

For lambdas with multiple return statements of *different* types, the compiler can't deduce the return type:

```
// error: return types must match if lambda has unspecified return type
auto l = [](int value) {
    if (value < 10) {
        return 1;
    } else {
        return 1.5;
    }
};
```

In this case you have to specify the return type explicitly:

```
// The return type is specified explicitly as 'double'
auto l = [](int value) -> double {
    if (value < 10) {
        return 1;
    } else {
        return 1.5;
    }
};
```

The rules for this match the rules for `auto` type deduction. Lambdas without explicitly specified return types never return references, so if a reference type is desired it must be explicitly specified as well:

```
auto copy = [](X& x) { return x; };       // 'copy' returns an X, so copies its input
auto ref  = [](X& x) -> X& { return x; }; // 'ref' returns an X&, no copy
```

Section 73.3: Capture by value

If you specify the variable's name in the capture list, the lambda will capture it by value. This means that the generated closure type for the lambda stores a copy of the variable. This also requires that the variable's type be *copy-constructible*:

```
int a = 0;

[a]() {
    return a;   // Ok, 'a' is captured by value
};
```

Version < C++14

```
auto p = std::unique_ptr<T>(...);

[p]() {         // Compile error; `unique_ptr` is not copy-constructible
    return p->createWidget();
};
```

From C++14 on, it is possible to initialize variables on the spot. This allows move only types to be captured in the lambda.

Version ≥ C++14

```
auto p = std::make_unique<T>(...);

[p = std::move(p)]() {
    return p->createWidget();
};
```

Even though a lambda captures variables by value when they are given by their name, such variables cannot be modified within the lambda body by default. This is because the closure type puts the lambda body in a declaration of `operator() const`.

The const applies to accesses to member variables of the closure type, and captured variables that are members of the closure (all appearances to the contrary):

```
int a = 0;

[a]() {
    a = 2;      // Illegal, 'a' is accessed via `const`

    decltype(a) a1 = 1;
    a1 = 2; // valid: variable 'a1' is not const
};
```

To remove the const, you have to specify the keyword mutable on the lambda:

```
int a = 0;

[a]() mutable {
    a = 2;      // OK, 'a' can be modified
    return a;
};
```

Because a was captured by value, any modifications done by calling the lambda will not affect a. The value of a was copied into the lambda when it was constructed, so the lambda's copy of a is separate from the external a variable.

```
int a = 5 ;
auto plus5Val = [a] (void) { return a + 5 ; } ;
auto plus5Ref = [&a] (void) {return a + 5 ; } ;

a = 7 ;
std::cout << a << ", value " << plus5Val() << ", reference " << plus5Ref() ;
// The result will be "7, value 10, reference 12"
```

Section 73.4: Recursive lambdas

Let's say we wish to write Euclid's gcd() as a lambda. As a function, it is:

```
int gcd(int a, int b) {
    return b == 0 ? a : gcd(b, a%b);
}
```

But a lambda cannot be recursive, it has no way to invoke itself. A lambda has no name and using this within the body of a lambda refers to a captured this (assuming the lambda is created in the body of a member function, otherwise it is an error). So how do we solve this problem?

Use std::function

We can have a lambda capture a reference to a not-yet constructed std::function:

```
std::function<int(int, int)> gcd = [&](int a, int b){
    return b == 0 ? a : gcd(b, a%b);
};
```

This works, but should be used sparingly. It's slow (we're using type erasure now instead of a direct function call), it's fragile (copying gcd or returning gcd will break since the lambda refers to the original object), and it won't work with generic lambdas.

Using two smart pointers:

```
auto gcd_self = std::make_shared<std::unique_ptr< std::function<int(int, int)> >>();
*gcd_self = std::make_unique<std::function<int(int, int)>>(
  [gcd_self](int a, int b){
    return b == 0 ? a : (**gcd_self)(b, a%b);
  };
};
```

This adds a lot of indirection (which is overhead), but it can be copied/returned, and all copies share state. It does let you return the lambda, and is otherwise less fragile than the above solution.

Use a Y-combinator

With the help of a short utility struct, we can solve all of these problems:

```
template <class F>
struct y_combinator {
    F f; // the lambda will be stored here

    // a forwarding operator():
    template <class... Args>
    decltype(auto) operator()(Args&&... args) const {
        // we pass ourselves to f, then the arguments.
        // the lambda should take the first argument as `auto&& recurse` or similar.
        return f(*this, std::forward<Args>(args)...);
    }
};
// helper function that deduces the type of the lambda:
template <class F>
y_combinator<std::decay_t<F>> make_y_combinator(F&& f) {
    return {std::forward<F>(f)};
}
// (Be aware that in C++17 we can do better than a `make_` function)
```

we can implement our `gcd` as:

```
auto gcd = make_y_combinator(
  [](auto&& gcd, int a, int b){
    return b == 0 ? a : gcd(b, a%b);
  }
);
```

The `y_combinator` is a concept from the lambda calculus that lets you have recursion without being able to name yourself until you are defined. This is exactly the problem lambdas have.

You create a lambda that takes "recurse" as its first argument. When you want to recurse, you pass the arguments to recurse.

The `y_combinator` then returns a function object that calls that function with its arguments, but with a suitable "recurse" object (namely the `y_combinator` itself) as its first argument. It forwards the rest of the arguments you call the `y_combinator` with to the lambda as well.

In short:

```
auto foo = make_y_combinator( [&](auto&& recurse, some arguments) {
  // write body that processes some arguments
  // when you want to recurse, call recurse(some other arguments)
```

```
});
```

and you have recursion in a lambda with no serious restrictions or significant overhead.

Section 73.5: Default capture

By default, local variables that are not explicitly specified in the capture list, cannot be accessed from within the lambda body. However, it is possible to implicitly capture variables named by the lambda body:

```
int a = 1;
int b = 2;

// Default capture by value
[=]() { return a + b; }; // OK; a and b are captured by value

// Default capture by reference
[&]() { return a + b; }; // OK; a and b are captured by reference
```

Explicit capturing can still be done alongside implicit default capturing. The explicit capture definition will override the default capture:

```
int a = 0;
int b = 1;

[=, &b]() {
    a = 2; // Illegal; 'a' is capture by value, and lambda is not 'mutable'
    b = 2; // OK; 'b' is captured by reference
};
```

Section 73.6: Class lambdas and capture of this

A lambda expression evaluated in a class' member function is implicitly a friend of that class:

```
class Foo
{
private:
    int i;

public:
    Foo(int val) : i(val) {}

    // definition of a member function
    void Test()
    {
        auto lamb = [](Foo &foo, int val)
        {
            // modification of a private member variable
            foo.i = val;
        };

        // lamb is allowed to access a private member, because it is a friend of Foo
        lamb(*this, 30);
    }
};
```

Such a lambda is not only a friend of that class, it has the same access as the class it is declared within has.

Lambdas can capture the `this` pointer which represents the object instance the outer function was called on. This

is done by adding `this` to the capture list:

```
class Foo
{
private:
    int i;

public:
    Foo(int val) : i(val) {}

    void Test()
    {
        // capture the this pointer by value
        auto lamb = [this](int val)
        {
            i = val;
        };

        lamb(30);
    }
};
```

When `this` is captured, the lambda can use member names of its containing class as though it were in its containing class. So an implicit `this->` is applied to such members.

Be aware that `this` is captured by value, but not the value of the type. It is captured by the value of `this`, which is a *pointer*. As such, the lambda does not *own* `this`. If the lambda out lives the lifetime of the object that created it, the lambda can become invalid.

This also means that the lambda can modify `this` without being declared `mutable`. It is the pointer which is `const`, not the object being pointed to. That is, unless the outer member function was itself a `const` function.

Also, be aware that the default capture clauses, both `[=]` and `[&]`, will *also* capture `this` implicitly. And they both capture it by the value of the pointer. Indeed, it is an error to specify `this` in the capture list when a default is given.

Version ≥ C++17

Lambdas can capture a copy of the `this` object, created at the time the lambda is created. This is done by adding `*this` to the capture list:

```
class Foo
{
private:
    int i;

public:
    Foo(int val) : i(val) {}

    void Test()
    {
        // capture a copy of the object given by the this pointer
        auto lamb = [*this](int val) mutable
        {
            i = val;
        };

        lamb(30); // does not change this->i
    }
};
```

Section 73.7: Capture by reference

If you precede a local variable's name with an `&`, then the variable will be captured by reference. Conceptually, this means that the lambda's closure type will have a reference variable, initialized as a reference to the corresponding variable from outside of the lambda's scope. Any use of the variable in the lambda body will refer to the original variable:

```
// Declare variable 'a'
int a = 0;

// Declare a lambda which captures 'a' by reference
auto set = [&a]() {
    a = 1;
};

set();
assert(a == 1);
```

The keyword `mutable` is not needed, because `a` itself is not `const`.

Of course, capturing by reference means that the lambda **must not** escape the scope of the variables it captures. So you could call functions that take a function, but you must not call a function that will *store* the lambda beyond the scope of your references. And you must not return the lambda.

Section 73.8: Generic lambdas

Version ≥ c++14

Lambda functions can take arguments of arbitrary types. This allows a lambda to be more generic:

```
auto twice = [](auto x){ return x+x; };

int i = twice(2); // i == 4
std::string s = twice("hello"); // s == "hellohello"
```

This is implemented in C++ by making the closure type's `operator()` overload a template function. The following type has equivalent behavior to the above lambda closure:

```
struct _unique_lambda_type
{
  template<typename T>
  auto operator() (T x) const {return x + x;}
};
```

Not all parameters in a generic lambda need be generic:

```
[](auto x, int y) {return x + y;}
```

Here, `x` is deduced based on the first function argument, while `y` will always be `int`.

Generic lambdas can take arguments by reference as well, using the usual rules for `auto` and `&`. If a generic parameter is taken as `auto&&`, this is a *forwarding* reference to the passed in argument and not an *rvalue* reference:

```
auto lamb1 = [](int &&x) {return x + 5;};
auto lamb2 = [](auto &&x) {return x + 5;};
int x = 10;
```

```
lamb1(x); // Illegal; must use `std::move(x)` for `int&&` parameters.
lamb2(x); // Legal; the type of `x` is deduced as `int&`.
```

Lambda functions can be variadic and perfectly forward their arguments:

```
auto lam = [](auto&&... args){return f(std::forward<decltype(args)>(args)...);};
```

or:

```
auto lam = [](auto&&... args){return f(decltype(args)(args)...);};
```

which only works "properly" with variables of type `auto&&`.

A strong reason to use generic lambdas is for visiting syntax.

```
boost::variant<int, double> value;
apply_visitor(value, [&](auto&& e){
  std::cout << e;
});
```

Here we are visiting in a polymorphic manner; but in other contexts, the names of the type we are passing isn't interesting:

```
mutex_wrapped<std::ostream&> os = std::cout;
os.write([&](auto&& os){
  os << "hello world\n";
});
```

Repeating the type of `std::ostream&` is noise here; it would be like having to mention the type of a variable every time you use it. Here we are creating a visitor, but no a polymorphic one; `auto` is used for the same reason you might use `auto` in a `for(:)` loop.

Section 73.9: Using lambdas for inline parameter pack unpacking

Version ≥ C++14

Parameter pack unpacking traditionally requires writing a helper function for each time you want to do it.

In this toy example:

```
template<std::size_t...Is>
void print_indexes( std::index_sequence<Is...> ) {
  using discard=int[];
  (void)discard{0,((void)(
    std::cout << Is << '\n' // here Is is a compile-time constant.
  ),0)...};
}
template<std::size_t I>
void print_indexes_upto() {
  return print_indexes( std::make_index_sequence<I>{} );
}
```

The `print_indexes_upto` wants to create and unpack a parameter pack of indexes. In order to do so, it must call a helper function. Every time you want to unpack a parameter pack you created, you end up having to create a

custom helper function to do it.

This can be avoided with lambdas.

You can unpack parameter packs into a set of invocations of a lambda, like this:

```
template<std::size_t I>
using index_t = std::integral_constant<std::size_t, I>;
template<std::size_t I>
constexpr index_t<I> index{};

template<class=void, std::size_t...Is>
auto index_over( std::index_sequence<Is...> ) {
  return [](auto&& f){
    using discard=int[];
    (void)discard{0,(void(
      f( index<Is> )
    ),0)...};
  };
}

template<std::size_t N>
auto index_over(index_t<N> = {}) {
  return index_over( std::make_index_sequence<N>{} );
}
```

Version ≥ C++17

With fold expressions, `index_over()` can be simplified to:

```
template<class=void, std::size_t...Is>
auto index_over( std::index_sequence<Is...> ) {
  return [](auto&& f){
    ((void)(f(index<Is>)), ...);
  };
}
```

Once you have done that, you can use this to replace having to manually unpack parameter packs with a second overload in other code, letting you unpack parameter packs "inline":

```
template<class Tup, class F>
void for_each_tuple_element(Tup&& tup, F&& f) {
  using T = std::remove_reference_t<Tup>;
  using std::tuple_size;
  auto from_zero_to_N = index_over< tuple_size<T>{} >();

  from_zero_to_N(
    [&](auto i){
      using std::get;
      f( get<i>( std::forward<Tup>(tup) ) );
    }
  );
}
```

The `auto i` passed to the lambda by the `index_over` is a `std::integral_constant<std::size_t, ???>`. This has a `constexpr` conversion to `std::size_t` that does not depend on the state of `this`, so we can use it as a compile-time constant, such as when we pass it to `std::get<i>` above.

To go back to the toy example at the top, rewrite it as:

```
template<std::size_t I>
void print_indexes_upto() {
  index_over(index<I>)([](auto i){
    std::cout << i << '\n'; // here i is a compile-time constant
  });
}
```

which is much shorter, and keeps logic in the code that uses it.

Live example to play with.

Section 73.10: Generalized capture

Version ≥ C++14

Lambdas can capture expressions, rather than just variables. This permits lambdas to store move-only types:

```
auto p = std::make_unique<T>(...);

auto lamb = [p = std::move(p)]() //Overrides capture-by-value of `p`.
{
  p->SomeFunc();
};
```

This moves the outer `p` variable into the lambda capture variable, also called `p`. `lamb` now owns the memory allocated by `make_unique`. Because the closure contains a type that is non-copyable, this means that `lamb` is itself non-copyable. But it can be moved:

```
auto lamb_copy = lamb; //Illegal
auto lamb_move = std::move(lamb); //legal.
```

Now `lamb_move` owns the memory.

Note that `std::function<>` requires that the values stored be copyable. You can write your own move-only-requiring `std::function`, or you could just stuff the lambda into a `shared_ptr` wrapper:

```
auto shared_lambda = [](auto&& f){
  return [spf = std::make_shared<std::decay_t<decltype(f)>>(decltype(f)(f))]
  (auto&&...args)->decltype(auto) {
    return (*spf)(decltype(args)(args)...);
  };
};
auto lamb_shared = shared_lambda(std::move(lamb_move));
```

takes our move-only lambda and stuffs its state into a shared pointer then returns a lambda that *can* be copied, and then stored in a `std::function` or similar.

Generalized capture uses `auto` type deduction for the variable's type. It will declare these captures as values by default, but they can be references as well:

```
int a = 0;

auto lamb = [&v = a](int add) //Note that `a` and `v` have different names
{
  v += add; //Modifies `a`
};
```

```
lamb(20); //`a` becomes 20.
```

Generalize capture does not need to capture an external variable at all. It can capture an arbitrary expression:

```
auto lamb = [p = std::make_unique<T>(...)]()
{
    p->SomeFunc();
}
```

This is useful for giving lambdas arbitrary values that they can hold and potentially modify, without having to declare them externally to the lambda. Of course, that is only useful if you do not intend to access those variables after the lambda has completed its work.

Section 73.11: Conversion to function pointer

If a lambda's capture list is empty, then the lambda has an implicit conversion to a function pointer that takes the same arguments and returns the same return type:

```
auto sorter = [](int lhs, int rhs) -> bool {return lhs < rhs;};

using func_ptr = bool(*)(int, int);
func_ptr sorter_func = sorter; // implicit conversion
```

Such a conversion may also be enforced using unary plus operator:

```
func_ptr sorter_func2 = +sorter; // enforce implicit conversion
```

Calling this function pointer behaves exactly like invoking `operator()` on the lambda. This function pointer is in no way reliant on the source lambda closure's existence. It therefore may outlive the lambda closure.

This feature is mainly useful for using lambdas with APIs that deal in function pointers, rather than C++ function objects.

Version ≥ C++14

Conversion to a function pointer is also possible for generic lambdas with an empty capture list. If necessary, template argument deduction will be used to select the correct specialization.

```
auto sorter = [](auto lhs, auto rhs) { return lhs < rhs; };
using func_ptr = bool(*)(int, int);
func_ptr sorter_func = sorter;  // deduces int, int
// note however that the following is ambiguous
// func_ptr sorter_func2 = +sorter;
```

Section 73.12: Porting lambda functions to C++03 using functors

Lambda functions in C++ are syntactic sugar that provide a very concise syntax for writing functors. As such, equivalent functionality can be obtained in C++03 (albeit much more verbose) by converting the lambda function into a functor:

```
// Some dummy types:
struct T1 {int dummy;};
struct T2 {int dummy;};
struct R {int dummy;};
```

```
// Code using a lambda function (requires C++11)
R use_lambda(T1 val, T2 ref) {
  // Use auto because the type of the lambda is unknown.
  auto lambda = [val, &ref](int arg1, int arg2) -> R {
    /* lambda-body */
    return R();
  };
  return lambda(12, 27);
}

// The functor class (valid C++03)
// Similar to what the compiler generates for the lambda function.
class Functor {
  // Capture list.
  T1 val;
  T2& ref;

public:
  // Constructor
  inline Functor(T1 val, T2& ref) : val(val), ref(ref) {}

  // Functor body
  R operator()(int arg1, int arg2) const {
    /* lambda-body */
    return R();
  }
};

// Equivalent to use_lambda, but uses a functor (valid C++03).
R use_functor(T1 val, T2 ref) {
  Functor functor(val, ref);
  return functor(12, 27);
}

// Make this a self-contained example.
int main() {
  T1 t1;
  T2 t2;
  use_functor(t1,t2);
  use_lambda(t1,t2);
  return 0;
}
```

If the lambda function is `mutable` then make the functor's call-operator non-const, i.e.:

```
R operator()(int arg1, int arg2) /*non-const*/ {
  /* lambda-body */
  return R();
}
```

Chapter 74: Value Categories

Section 74.1: Value Category Meanings

Expressions in C++ are assigned a particular value category, based on the result of those expressions. Value categories for expressions can affect C++ function overload resolution.

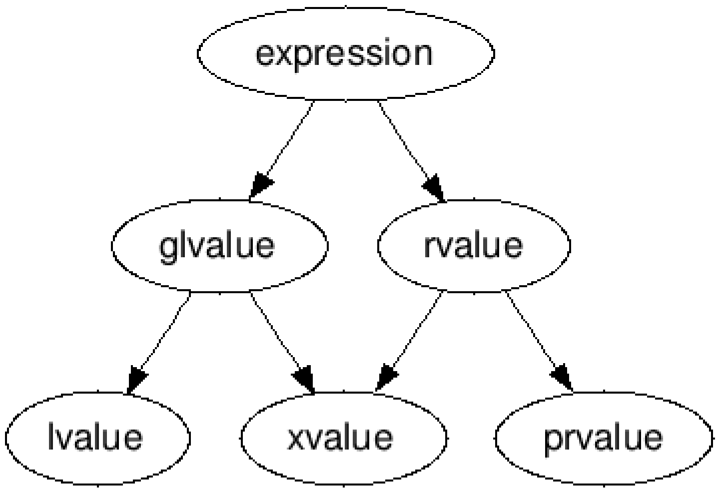
Value categories determines two important-but-separate properties about an expression. One property is whether the expression has identity. An expression has identity if it refers to an object that has a variable name. The variable name may not be involved in the expression, but the object can still have one.

The other property is whether it is legal to implicitly move from the expression's value. Or more specifically, whether the expression, when used as a function parameter, will bind to r-value parameter types or not.

C++ defines 3 value categories which represent the useful combination of these properties: lvalue (expressions with identity but not movable from), xvalue (expressions with identity that are moveable from), and prvalue (expressions without identity that are moveable from). C++ does not have expressions which have no identity and cannot be moved from.

C++ defines two other value categories, each based solely on one of these properties: glvalue (expressions with identity) and rvalue (expressions that can be moved from). These act as useful groupings of the prior categories.

This graph serves as an illustration:



Section 74.2: rvalue

An rvalue expression is any expression which can be implicitly moved from, regardless of whether it has identity.

More precisely, rvalue expressions may be used as the argument to a function that takes a parameter of type `T &&` (where `T` is the type of `expr`). *Only* rvalue expressions may be given as arguments to such function parameters; if a non-rvalue expression is used, then overload resolution will pick any function that does not use an rvalue reference parameter. And if none exist, then you get an error.

The category of rvalue expressions includes all xvalue and prvalue expressions, and only those expressions.

The standard library function `std::move` exists to explicitly transform a non-rvalue expression into an rvalue. More specifically, it turns the expression into an xvalue, since even if it was an identity-less prvalue expression before, by passing it as a parameter to `std::move`, it gains identity (the function's parameter name) and becomes an xvalue.

Consider the following:

```
std::string str("init");                       //1
std::string test1(str);                        //2
std::string test2(std::move(str));             //3

str = std::string("new value");                //4
std::string &&str_ref = std::move(str);        //5
std::string test3(str_ref);                    //6
```

`std::string` has a constructor which takes a single parameter of type `std::string&&`, commonly called a "move constructor". However, the value category of the expression `str` is not an rvalue (specifically it is an lvalue), so it cannot call that constructor overload. Instead, it calls the `const std::string&` overload, the copy constructor.

Line 3 changes things. The return value of `std::move` is a `T&&`, where `T` is the base type of the parameter passed in. So `std::move(str)` returns `std::string&&`. A function call who's return value is an rvalue reference is an rvalue expression (specifically an xvalue), so it may call the move constructor of `std::string`. After line 3, `str` has been moved from (who's contents are now undefined).

Line 4 passes a temporary to the assignment operator of `std::string`. This has an overload which takes a `std::string&&`. The expression `std::string("new value")` is an rvalue expression (specifically a prvalue), so it may call that overload. Thus, the temporary is moved into `str`, replacing the undefined contents with specific contents.

Line 5 creates a named rvalue reference called `str_ref` that refers to `str`. This is where value categories get confusing.

See, while `str_ref` is an rvalue reference to `std::string`, the value category of the expression `str_ref` *is not an rvalue*. It is an lvalue expression. Yes, really. Because of this, one cannot call the move constructor of `std::string` with the expression `str_ref`. Line 6 therefore *copies* the value of `str` into `test3`.

To move it, we would have to employ `std::move` again.

Section 74.3: xvalue

An xvalue (eXpiring value) expression is an expression which has identity and represents an object which can be implicitly moved from. The general idea with xvalue expressions is that the object they represent is going to be destroyed soon (hence the "eXpiring" part), and therefore implicitly moving from them is fine.

Given:

```
struct X { int n; };
extern X x;

4;                     // prvalue: does not have an identity
x;                     // lvalue
x.n;                   // lvalue
std::move(x);          // xvalue
std::forward<X&>(x);   // lvalue
X{4};                  // prvalue: does not have an identity
X{4}.n;                // xvalue: does have an identity and denotes resources
                       // that can be reused
```

Section 74.4: prvalue

A prvalue (pure-rvalue) expression is an expression which lacks identity, whose evaluation is typically used to

initialize an object, and which can be implicitly moved from. These include, but are not limited to:

- Expressions that represent temporary objects, such as `std::string("123")`.
- A function call expression that does not return a reference
- A literal (*except* a string literal - those are lvalues), such has `1`, `true`, `0.5f`, or `'a'`
- A lambda expression

The built-in addressof operator (`&`) cannot be applied on these expressions.

Section 74.5: lvalue

An lvalue expression is an expression which has identity, but cannot be implicitly moved from. Among these are expressions that consist of a variable name, function name, expressions that are built-in dereference operator uses and expressions that refer to lvalue references.

The typical lvalue is simply a name, but lvalues can come in other flavors as well:

```
struct X { ... };

X x;         // x is an lvalue
X* px = &x;  // px is an lvalue
*px = X{};   // *px is also an lvalue, X{} is a prvalue

X* foo_ptr();  // foo_ptr() is a prvalue
X& foo_ref();  // foo_ref() is an lvalue
```

Additionally, while most literals (e.g. `4`, `'x'`, etc.) are prvalues, string literals are lvalues.

Section 74.6: glvalue

A glvalue (a "generalized lvalue") expression is any expression which has identity, regardless of whether it can be moved from or not. This category includes lvalues (expressions that have identity but can't be moved from) and xvalues (expressions that have identity, and can be moved from), but excludes prvalues (expressions without identity).

If an expression has a *name*, it's a glvalue:

```
struct X { int n; };
X foo();

X x;
x; // has a name, so it's a glvalue
std::move(x); // has a name (we're moving from "x"), so it's a glvalue
              // can be moved from, so it's an xvalue not an lvalue

foo(); // has no name, so is a prvalue, not a glvalue
X{};   // temporary has no name, so is a prvalue, not a glvalue
X{}.n; // HAS a name, so is a glvalue. can be moved from, so it's an xvalue
```

Chapter 75: Preprocessor

The C preprocessor is a simple text parser/replacer that is run before the actual compilation of the code. Used to extend and ease the use of the C (and later C++) language, it can be used for:

a. **Including other files** using `#include`

b. **Define a text-replacement macro** using `#define`

c. **Conditional Compilation** using`#if` `#ifdef`

d. **Platform/Compiler specific logic** (as an extension of conditional compilation)

Section 75.1: Include Guards

A header file may be included by other header files. A source file (compilation unit) that includes multiple headers may therefore, indirectly, include some headers more than once. If such a header file that is included more than once contains definitions, the compiler (after preprocessing) detects a violation of the One Definition Rule (e.g. §3.2 of the 2003 C++ standard) and therefore issues a diagnostic and compilation fails.

Multiple inclusion is prevented using "include guards", which are sometimes also known as header guards or macro guards. These are implemented using the preprocessor `#define`, `#ifndef`, `#endif` directives.

e.g.

```
// Foo.h
#ifndef FOO_H_INCLUDED
#define FOO_H_INCLUDED

class Foo    //  a class definition
{
};

#endif
```

The key advantage of using include guards is that they will work with all standard-compliant compilers and preprocessors.

However, include guards also cause some problems for developers, as it is necessary to ensure the macros are unique within all headers used in a project. Specifically, if two (or more) headers use `FOO_H_INCLUDED` as their include guard, the first of those headers included in a compilation unit will effectively prevent the others from being included. Particular challenges are introduced if a project uses a number of third-party libraries with header files that happen to use include guards in common.

It is also necessary to ensure that the macros used in include guards do not conflict with any other macros defined in header files.

Most C++ implementations also support the `#pragma once` directive which ensures the file is only included once within a single compilation. This is a *de facto* standard directive, but it is not part of any ISO C++ standard. For example:

```
// Foo.h
#pragma once

class Foo
```

```
{
};
```

While `#pragma once` avoids some problems associated with include guards, a `#pragma` - by definition in the standards - is inherently a compiler-specific hook, and will be silently ignored by compilers that don't support it. Projects which use `#pragma once` are more difficult to port to compilers that don't support it.

A number of coding guidelines and assurance standards for C++ specifically discourage any use of the preprocessor other than to `#include` header files or for the purposes of placing include guards in headers.

Section 75.2: Conditional logic and cross-platform handling

In a nutshell, conditional pre-processing logic is about making code-logic available or unavailable for compilation using macro definitions.

Three prominent use-cases are:

- different **app profiles** (e.g. debug, release, testing, optimised) that can be candidates of the same app (e.g. with extra logging).
- **cross-platform compiles** - single code-base, multiple compilation platforms.
- utilising a common code-base for multiple **application versions** (e.g. Basic, Premium and Pro versions of a software) - with slightly different features.

Example a: A cross-platform approach for removing files (illustrative):

```
#ifdef _WIN32
#include <windows.h> // and other windows system files
#endif
#include <cstdio>

bool remove_file(const std::string &path)
{
#ifdef _WIN32
  return DeleteFile(path.c_str());
#elif defined(_POSIX_VERSION) || defined(__unix__)
  return (0 == remove(path.c_str()));
#elif defined(__APPLE__)
  //TODO: check if NSAPI has a more specific function with permission dialog
  return (0 == remove(path.c_str()));
#else
#error "This platform is not supported"
#endif
}
```

Macros like `_WIN32`, `__APPLE__` or `__unix__` are normally predefined by corresponding implementations.

Example b: Enabling additional logging for a debug build:

```
void s_PrintAppStateOnUserPrompt()
{
    std::cout << "--------BEGIN-DUMP---------------\n"
              << AppState::Instance()->Settings().ToString() << "\n"
#if ( 1 == TESTING_MODE ) //privacy: we want user details only when testing
              << ListToString(AppState::UndoStack()->GetActionNames())
              << AppState::Instance()->CrntDocument().Name()
              << AppState::Instance()->CrntDocument().SignatureSHA() << "\n"
#endif
```

```
                << "--------END-DUMP---------------\n"
}
```

Example c: Enable a premium feature in a separate product build (note: this is illustrative. it is often a better idea to allow a feature to be unlocked without the need to reinstall an application)

```
void MainWindow::OnProcessButtonClick()
{
#ifndef _PREMIUM
    CreatePurchaseDialog("Buy App Premium", "This feature is available for our App Premium users.
Click the Buy button to purchase the Premium version at our website");
    return;
#endif
    //...actual feature logic here
}
```

Some common tricks:

Defining symbols at invocation time:

The preprocessor can be called with predefined symbols (with optional initialisation). For example this command (`gcc -E` runs only the preprocessor)

```
gcc -E -DOPTIMISE_FOR_OS_X -DTESTING_MODE=1 Sample.cpp
```

processes Sample.cpp in the same way as it would if `#define OPTIMISE_FOR_OS_X` and `#define TESTING_MODE 1` were added to the top of Sample.cpp.

Ensuring a macro is defined:

If a macro isn't defined and its value is compared or checked, the preprocessor almost always silently assumes the value to be `0`. There are a few ways to work with this. One approach is to assume that the default settings are represented as 0, and any changes (e.g. to the app build profile) needs to be explicitly done (e.g. ENABLE_EXTRA_DEBUGGING=0 by default, set -DENABLE_EXTRA_DEBUGGING=1 to override). Another approach is make all definitions and defaults explicit. This can be achieved using a combination of `#ifndef` and `#error` directives:

```
#ifndef (ENABLE_EXTRA_DEBUGGING)
// please include DefaultDefines.h if not already included.
#    error "ENABLE_EXTRA_DEBUGGING is not defined"
#else
#    if ( 1 == ENABLE_EXTRA_DEBUGGING )
  //code
#    endif
#endif
```

Section 75.3: X-macros

An idiomatic technique for generating repeating code structures at compile time.

An X-macro consists of two parts: the list, and the execution of the list.

Example:

```
#define LIST \
    X(dog)   \
```

```
    X(cat)   \
    X(racoon)

// class Animal {
//  public:
//    void say();
// };

#define X(name) Animal name;
LIST
#undef X

int main() {
#define X(name) name.say();
    LIST
#undef X

    return 0;
}
```

which is expanded by the preprocessor into the following:

```
Animal dog;
Animal cat;
Animal racoon;

int main() {
    dog.say();
    cat.say();
    racoon.say();

    return 0;
}
```

As lists become bigger (let's say, more than 100 elements), this technique helps to avoid excessive copy-pasting.

Source: https://en.wikipedia.org/wiki/X_Macro

See also: X-macros

If defining a seamingly irrelevant `X` before using `LIST` is not to your liking, you can pass a macro name as an argument as well:

```
#define LIST(MACRO) \
    MACRO(dog) \
    MACRO(cat) \
    MACRO(racoon)
```

Now, you explicitly specify which macro should be used when expanding the list, e.g.

```
#define FORWARD_DECLARE_ANIMAL(name) Animal name;
LIST(FORWARD_DECLARE_ANIMAL)
```

If each invocation of the `MACRO` should take additional parameters - constant with respect to the list, variadic macros can be used

```
//a walkaround for Visual studio
#define EXPAND(x) x
```

```
#define LIST(MACRO, ...) \
    EXPAND(MACRO(dog, __VA_ARGS__)) \
    EXPAND(MACRO(cat, __VA_ARGS__)) \
    EXPAND(MACRO(racoon, __VA_ARGS__))
```

The first argument is supplied by the `LIST`, while the rest is provided by the user in the `LIST` invocation. For example:

```
#define FORWARD_DECLARE(name, type, prefix) type prefix##name;
LIST(FORWARD_DECLARE,Animal,anim_)
LIST(FORWARD_DECLARE,Object,obj_)
```

will expand to

```
Animal anim_dog;
Animal anim_cat;
Animal anim_racoon;
Object obj_dog;
Object obj_cat;
Object obj_racoon;
```

Section 75.4: Macros

Macros are categorized into two main groups: object-like macros and function-like macros. Macros are treated as a token substitution early in the compilation process. This means that large (or repeating) sections of code can be abstracted into a preprocessor macro.

```
// This is an object-like macro
#define    PI         3.14159265358979

// This is a function-like macro.
// Note that we can use previously defined macros
// in other macro definitions (object-like or function-like)
// But watch out, its quite useful if you know what you're doing, but the
// Compiler doesn't know which type to handle, so using inline functions instead
// is quite recommended (But e.g. for Minimum/Maximum functions it is quite useful)
#define    AREA(r)    (PI*(r)*(r))

// They can be used like this:
double pi_macro   = PI;
double area_macro = AREA(4.6);
```

The Qt library makes use of this technique to create a meta-object system by having the user declare the Q_OBJECT macro at the head of the user-defined class extending QObject.

Macro names are usually written in all caps, to make them easier to differentiate from normal code. This isn't a requirement, but is merely considered good style by many programmers.

When an object-like macro is encountered, it's expanded as a simple copy-paste operation, with the macro's name being replaced with its definition. When a function-like macro is encountered, both its name and its parameters are expanded.

```
double pi_squared = PI * PI;
// Compiler sees:
double pi_squared = 3.14159265358979 * 3.14159265358979;

double area = AREA(5);
```

```
// Compiler sees:
double area = (3.14159265358979*(5)*(5))
```

Due to this, function-like macro parameters are often enclosed within parentheses, as in `AREA()` above. This is to prevent any bugs that can occur during macro expansion, specifically bugs caused by a single macro parameter being composed of multiple actual values.

```
#define BAD_AREA(r) PI * r * r

double bad_area = BAD_AREA(5 + 1.6);
// Compiler sees:
double bad_area = 3.14159265358979 * 5 + 1.6 * 5 + 1.6;

double good_area = AREA(5 + 1.6);
// Compiler sees:
double good_area = (3.14159265358979*(5 + 1.6)*(5 + 1.6));
```

Also note that due to this simple expansion, care must be taken with the parameters passed to macros, to prevent unexpected side effects. If the parameter is modified during evaluation, it will be modified each time it is used in the expanded macro, which usually isn't what we want. This is true even if the macro encloses the parameters in parentheses to prevent expansion from breaking anything.

```
int oops = 5;
double incremental_damage = AREA(oops++);
// Compiler sees:
double incremental_damage = (3.14159265358979*(oops++)*(oops++));
```

Additionally, macros provide no type-safety, leading to hard-to-understand errors about type mismatch.

As programmers normally terminate lines with a semicolon, macros that are intended to be used as standalone lines are often designed to "swallow" a semicolon; this prevents any unintended bugs from being caused by an extra semicolon.

```
#define IF_BREAKER(Func) Func();

if (some_condition)
    // Oops.
    IF_BREAKER(some_func);
else
    std::cout << "I am accidentally an orphan." << std::endl;
```

In this example, the inadvertent double semicolon breaks the `if...else` block, preventing the compiler from matching the `else` to the `if`. To prevent this, the semicolon is omitted from the macro definition, which will cause it to "swallow" the semicolon immediately following any usage of it.

```
#define IF_FIXER(Func) Func()

if (some_condition)
    IF_FIXER(some_func);
else
    std::cout << "Hooray!  I work again!" << std::endl;
```

Leaving off the trailing semicolon also allows the macro to be used without ending the current statement, which can be beneficial.

```
#define DO_SOMETHING(Func, Param) Func(Param, 2)
```

```
// ...

some_function(DO_SOMETHING(some_func, 3), DO_SOMETHING(some_func, 42));
```

Normally, a macro definition ends at the end of the line. If a macro needs to cover multiple lines, however, a backslash can be used at the end of a line to indicate this. This backslash must be the last character in the line, which indicates to the preprocessor that the following line should be concatenated onto the current line, treating them as a single line. This can be used multiple times in a row.

```
#define TEXT "I \
am \
many \
lines."

// ...

std::cout << TEXT << std::endl; // Output:   I am many lines.
```

This is especially useful in complex function-like macros, which may need to cover multiple lines.

```
#define CREATE_OUTPUT_AND_DELETE(Str) \
    std::string* tmp = new std::string(Str); \
    std::cout << *tmp << std::endl; \
    delete tmp;

// ...

CREATE_OUTPUT_AND_DELETE("There's no real need for this to use 'new'.")
```

In the case of more complex function-like macros, it can be useful to give them their own scope to prevent possible name collisions or to cause objects to be destroyed at the end of the macro, similar to an actual function. A common idiom for this is *do while 0*, where the macro is enclosed in a *do-while* block. This block is generally *not* followed with a semicolon, allowing it to swallow a semicolon.

```
#define DO_STUFF(Type, Param, ReturnVar) do { \
    Type temp(some_setup_values); \
    ReturnVar = temp.process(Param); \
} while (0)

int x;
DO_STUFF(MyClass, 41153.7, x);

// Compiler sees:

int x;
do {
    MyClass temp(some_setup_values);
    x = temp.process(41153.7);
} while (0);
```

There are also variadic macros; similarly to variadic functions, these take a variable number of arguments, and then expand them all in place of a special "Varargs" parameter, `__VA_ARGS__`.

```
#define VARIADIC(Param, ...) Param(__VA_ARGS__)

VARIADIC(printf, "%d", 8);
// Compiler sees:
```

```
printf("%d", 8);
```

Note that during expansion, `__VA_ARGS__` can be placed anywhere in the definition, and will be expanded correctly.

```
#define VARIADIC2(POne, PTwo, PThree, ...) POne(PThree, __VA_ARGS__, PTwo)

VARIADIC2(some_func, 3, 8, 6, 9);
// Compiler sees:
some_func(8, 6, 9, 3);
```

In the case of a zero-argument variadic parameter, different compilers will handle the trailing comma differently. Some compilers, such as Visual Studio, will silently swallow the comma without any special syntax. Other compilers, such as GCC, require you to place ## immediately before `__VA_ARGS__`. Due to this, it is wise to conditionally define variadic macros when portability is a concern.

```
// In this example, COMPILER is a user-defined macro specifying the compiler being used.

#if       COMPILER == "VS"
    #define VARIADIC3(Name, Param, ...) Name(Param, __VA_ARGS__)
#elif     COMPILER == "GCC"
    #define VARIADIC3(Name, Param, ...) Name(Param, ##__VA_ARGS__)
#endif /* COMPILER */
```

Section 75.5: Predefined macros

Predefined macros are those that the compiler defines (in contrast to those user defines in the source file). Those macros must not be re-defined or undefined by user.

The following macros are predefined by the C++ standard:

- `__LINE__` contains the line number of the line this macro is used on, and can be changed by the `#line` directive.
- `__FILE__` contains the filename of the file this macro is used in, and can be changed by the `#line` directive.
- `__DATE__` contains date (in `"Mmm dd yyyy"` format) of the file compilation, where *Mmm* is formatted as if obtained by a call to `std::asctime()`.
- `__TIME__` contains time (in `"hh:mm:ss"` format) of the file compilation.
- `__cplusplus` is defined by (conformant) C++ compilers while compiling C++ files. Its value is the standard version the compiler is **fully** conformant with, i.e. `199711L` for C++98 and C++03, `201103L` for C++11 and `201402L` for C++14 standard.

Version ≥ c++11

- `__STDC_HOSTED__` is defined to `1` if the implementation is *hosted*, or `0` if it is *freestanding*.

Version ≥ c++17

- `__STDCPP_DEFAULT_NEW_ALIGNMENT__` contains a `size_t` literal, which is the alignment used for a call to alignment-unaware `operator new`.

Additionally, the following macros are allowed to be predefined by implementations, and may or may not be present:

- `__STDC__` has implementation-dependent meaning, and is usually defined only when compiling a file as C, to signify full C standard compliance. (Or never, if the compiler decides not to support this macro.)

Version ≥ c++11

- `__STDC_VERSION__` has implementation-dependent meaning, and its value is usually the C version, similarly to how `__cplusplus` is the C++ version. (Or is not even defined, if the compiler decides not to support this macro.)
- `__STDC_MB_MIGHT_NEQ_WC__` is defined to `1`, if values of the narrow encoding of the basic character set might not be equal to the values of their wide counterparts (e.g. if `(uintmax_t)'x' != (uintmax_t)L'x'`)
- `__STDC_ISO_10646__` is defined if `wchar_t` is encoded as Unicode, and expands to an integer constant in the form `yyyymmL`, indicating the latest Unicode revision supported.
- `__STDCPP_STRICT_POINTER_SAFETY__` is defined to `1`, if the implementation has *strict pointer safety* (otherwise it has *relaxed pointer safety*)
- `__STDCPP_THREADS__` is defined to `1`, if the program can have more than one thread of execution (applicable to *freestanding implementation* — *hosted implementations* can always have more than one thread)

It is also worth mentioning `__func__`, which is not an macro, but a predefined function-local variable. It contains the name of the function it is used in, as a static character array in an implementation-defined format.

On top of those standard predefined macros, compilers can have their own set of predefined macros. One must refer to the compiler documentation to learn those. E.g.:

- gcc
- Microsoft Visual C++
- clang
- Intel C++ Compiler

Some of the macros are just to query support of some feature:

```
#ifdef __cplusplus // if compiled by C++ compiler
extern "C"{ // C code has to be decorated
   // C library header declarations here
}
#endif
```

Others are very useful for debugging:

Version ≥ c++11

```
bool success = doSomething( /*some arguments*/ );
if( !success ){
    std::cerr << "ERROR: doSomething() failed on line " << __LINE__ - 2
              << " in function " << __func__ << "()"
              << " in file " << __FILE__
              << std::endl;
}
```

And others for trivial version control:

```
int main( int argc, char *argv[] ){
    if( argc == 2 && std::string( argv[1] ) == "-v" ){
        std::cout << "Hello World program\n"
                  << "v 1.1\n" // I have to remember to update this manually
                  << "compiled: " << __DATE__ << ' ' << __TIME__ // this updates automagically
                  << std::endl;
    }
    else{
        std::cout << "Hello World!\n";
    }
}
```

Section 75.6: Preprocessor Operators

operator or stringizing operator is used to convert a Macro parameter to a string literal. It can only be used with the Macros having arguments.

```
// preprocessor will convert the parameter x to the string literal x
#define PRINT(x) printf(#x "\n")

PRINT(This line will be converted to string by preprocessor);
// Compiler sees
printf("This line will be converted to string by preprocessor""\n");
```

Compiler concatenate two strings and the final `printf()` argument will be a string literal with newline character at its end.

Preprocessor will ignore the spaces before or after the macro argument. So below print statement will give us the same result.

```
PRINT(   This line will be converted to string by preprocessor );
```

If the parameter of the string literal requires an escape sequence like before a double quote() it will automatically be inserted by the preprocessor.

```
PRINT(This "line" will be converted to "string" by preprocessor);
// Compiler sees
printf("This \"line\" will be converted to \"string\" by preprocessor""\n");
```

operator or Token pasting operator is used to concatenate two parameters or tokens of a Macro.

```
// preprocessor will combine the variable and the x
#define PRINT(x) printf("variable" #x " = %d", variable##x)

int variableY = 15;
PRINT(Y);
//compiler sees
printf("variable""Y"" = %d", variableY);
```

and the final output will be

```
variableY = 15
```

Section 75.7: #pragma once

Most, but not all, C++ implementations support the `#pragma once` directive which ensures the file is only included once within a single compilation. It is not part of any ISO C++ standard. For example:

```
// Foo.h
#pragma once

class Foo
{
};
```

While `#pragma once` avoids some problems associated with include guards, a `#pragma` - by definition in the standards - is inherently a compiler-specific hook, and will be silently ignored by compilers that don't support it.

Projects which use `#pragma once` must be modified to be standard-compliant.

With some compilers - particularly those that employ precompiled headers - `#pragma once` can result in a considerable speedup of the compilation process. Similarly, some preprocessors achieve speedup of compilation by tracking which headers have employed include guards. The net benefit, when both `#pragma once` and include guards are employed, depends on the implementation and can be either an increase or decrease of compilation times.

`#pragma once` combined with include guards was the recommended layout for header files when writing MFC based applications on windows, and was generated by Visual Studio's `add class`, `add dialog`, `add windows` wizards. Hence it is very common to find them combined in C++ Windows Applicants.

Section 75.8: Preprocessor error messages

Compile errors can be generated using the preprocessor. This is useful for a number of reasons some of which include, notifying a user if they are on an unsupported platform or an unsupported compiler.

e.g. Return Error if gcc version is 3.0.0 or earlier.

```
#if __GNUC__ < 3
#error "This code requires gcc > 3.0.0"
#endif
```

e.g. Return Error if compiling on an Apple computer.

```
#ifdef __APPLE__
#error "Apple products are not supported in this release"
#endif
```

Chapter 76: Data Structures in C++

Section 76.1: Linked List implementation in C++

Creating a List Node

```
class listNode
{
    public:
    int data;
    listNode *next;
    listNode(int val):data(val),next(NULL){}
};
```

Creating List class

```
class List
{
    public:
    listNode *head;
    List():head(NULL){}
    void insertAtBegin(int val);
    void insertAtEnd(int val);
    void insertAtPos(int val);
    void remove(int val);
    void print();
    ~List();
};
```

Insert a new node at the beginning of the list

```
void List::insertAtBegin(int val)//inserting at front of list
{
    listNode *newnode = new listNode(val);
    newnode->next=this->head;
    this->head=newnode;
}
```

Insert a new node at the end of the list

```
void List::insertAtEnd(int val) //inserting at end of list
{
    if(head==NULL)
    {
        insertAtBegin(val);
        return;
    }
    listNode *newnode = new listNode(val);
    listNode *ptr=this->head;
    while(ptr->next!=NULL)
    {
        ptr=ptr->next;
    }
    ptr->next=newnode;
}
```

Insert at a particular position in list

```
void List::insertAtPos(int pos,int val)
{
    listNode *newnode=new listNode(val);
    if(pos==1)
    {
        //as head
        newnode->next=this->head;
        this->head=newnode;
        return;
    }
    pos--;
    listNode *ptr=this->head;
    while(ptr!=NULL && --pos)
    {
        ptr=ptr->next;
    }
    if(ptr==NULL)
    return;//not enough elements
    newnode->next=ptr->next;
    ptr->next=newnode;
}
```

Removing a node from the list

```
void List::remove(int toBeRemoved)//removing an element
{
    if(this->head==NULL)
    return; //empty
    if(this->head->data==toBeRemoved)
    {
        //first node to be removed
        listNode *temp=this->head;
        this->head=this->head->next;
        delete(temp);
        return;
    }
    listNode *ptr=this->head;
    while(ptr->next!=NULL && ptr->next->data!=toBeRemoved)
    ptr=ptr->next;
    if(ptr->next==NULL)
    return;//not found
    listNode *temp=ptr->next;
    ptr->next=ptr->next->next;
    delete(temp);
}
```

Print the list

```
void List::print()//printing the list
{
    listNode *ptr=this->head;
    while(ptr!=NULL)
    {
        cout<<ptr->data<<" " ;
        ptr=ptr->next;
    }
    cout<<endl;
}
```

Destructor for the list

```
List::~List()
{
    listNode *ptr=this->head,*next=NULL;
    while(ptr!=NULL)
    {
        next=ptr->next;
        delete(ptr);
        ptr=next;
    }
}
```

Chapter 77: Templates

Classes, functions, and (since C++14) variables can be templated. A template is a piece of code with some free parameters that will become a concrete class, function, or variable when all parameters are specified. Parameters can be types, values, or themselves templates. A well-known template is `std::vector`, which becomes a concrete container type when the element type is specified, *e.g.*, `std::vector<int>`.

Section 77.1: Basic Class Template

The basic idea of a class template is that the template parameter gets substituted by a type at compile time. The result is that the same class can be reused for multiple types. The user specifies which type will be used when a variable of the class is declared. Three examples of this are shown in `main()`:

```
#include <iostream>
using std::cout;

template <typename T>         // A simple class to hold one number of any type
class Number {
public:
    void setNum(T n);         // Sets the class field to the given number
    T plus1() const;          // returns class field's "follower"
private:
    T num;                    // Class field
};

template <typename T>         // Set the class field to the given number
void Number<T>::setNum(T n) {
    num = n;
}

template <typename T>         // returns class field's "follower"
T Number<T>::plus1() const {
    return num + 1;
}

int main() {
    Number<int> anInt;        // Test with an integer (int replaces T in the class)
    anInt.setNum(1);
    cout << "My integer + 1 is " << anInt.plus1() << "\n";     // Prints 2

    Number<double> aDouble;   // Test with a double
    aDouble.setNum(3.1415926535897);
    cout << "My double + 1 is " << aDouble.plus1() << "\n";    // Prints 4.14159

    Number<float> aFloat;     // Test with a float
    aFloat.setNum(1.4);
    cout << "My float + 1 is " << aFloat.plus1() << "\n";      // Prints 2.4

    return 0;  // Successful completion
}
```

Section 77.2: Function Templates

Templating can also be applied to functions (as well as the more traditional structures) with the same effect.

```
// 'T' stands for the unknown type
// Both of our arguments will be of the same type.
```

```cpp
template<typename T>
void printSum(T add1, T add2)
{
    std::cout << (add1 + add2) << std::endl;
}
```

This can then be used in the same way as structure templates.

```cpp
printSum<int>(4, 5);
printSum<float>(4.5f, 8.9f);
```

In both these case the template argument is used to replace the types of the parameters; the result works just like a normal C++ function (if the parameters don't match the template type the compiler applies the standard conversions).

One additional property of template functions (unlike template classes) is that the compiler can infer the template parameters based on the parameters passed to the function.

```cpp
printSum(4, 5);     // Both parameters are int.
                    // This allows the compiler deduce that the type
                    // T is also int.

printSum(5.0, 4);   // In this case the parameters are two different types.
                    // The compiler is unable to deduce the type of T
                    // because there are contradictions. As a result
                    // this is a compile time error.
```

This feature allows us to simplify code when we combine template structures and functions. There is a common pattern in the standard library that allows us to make `template structure X` using a helper function `make_X()`.

```cpp
// The make_X pattern looks like this.
// 1) A template structure with 1 or more template types.
template<typename T1, typename T2>
struct MyPair
{
    T1      first;
    T2      second;
};
// 2) A make function that has a parameter type for
//    each template parameter in the template structure.
template<typename T1, typename T2>
MyPair<T1, T2> make_MyPair(T1 t1, T2 t2)
{
    return MyPair<T1, T2>{t1, t2};
}
```

How does this help?

```cpp
auto val1 = MyPair<int, float>{5, 8.7};     // Create object explicitly defining the types
auto val2 = make_MyPair(5, 8.7);            // Create object using the types of the parameters.
                                            // In this code both val1 and val2 are the same
                                            // type.
```

Note: This is not designed to shorten the code. This is designed to make the code more robust. It allows the types to be changed by changing the code in a single place rather than in multiple locations.

Section 77.3: Variadic template data structures

Version ≥ C++14

It is often useful to define classes or structures that have a variable number and type of data members which are defined at compile time. The canonical example is `std::tuple`, but sometimes is it is necessary to define your own custom structures. Here is an example that defines the structure using compounding (rather than inheritance as with `std::tuple`. Start with the general (empty) definition, which also serves as the base-case for recrusion termination in the later specialisation:

```
template<typename ... T>
struct DataStructure {};
```

This already allows us to define an empty structure, `DataStructure<> data`, albeit that isn't very useful yet.

Next comes the recursive case specialisation:

```
template<typename T, typename ... Rest>
struct DataStructure<T, Rest ...>
{
    DataStructure(const T& first, const Rest& ... rest)
        : first(first)
        , rest(rest...)
    {}

    T first;
    DataStructure<Rest ... > rest;
};
```

This is now sufficient for us to create arbitrary data structures, like `DataStructure<int, float, std::string> data(1, 2.1, "hello")`.

So what's going on? First, note that this is a specialisation whose requirement is that at least one variadic template parameter (namely `T` above) exists, whilst not caring about the specific makeup of the pack `Rest`. Knowing that `T` exists allows the definition of its data member, `first`. The rest of the data is recursively packaged as `DataStructure<Rest ... > rest`. The constructor initiates both of those members, including a recursive constructor call to the `rest` member.

To understand this better, we can work through an example: suppose you have a declaration `DataStructure<int, float> data`. The declaration first matches against the specialisation, yielding a structure with `int first` and `DataStructure<float> rest` data members. The `rest` definition again matches this specialisation, creating its own `float first` and `DataStructure<> rest` members. Finally this last `rest` matches against the base-case defintion, producing an empty structure.

You can visualise this as follows:

```
DataStructure<int, float>
   -> int first
   -> DataStructure<float> rest
         -> float first
         -> DataStructure<> rest
              -> (empty)
```

Now we have the data structure, but its not terribly useful yet as we cannot easily access the individual data elements (for example to access the last member of `DataStructure<int, float, std::string> data` we would have to use `data.rest.rest.first`, which is not exactly user-friendly). So we add a `get` method to it (only needed

in the specialisation as the base-case structure has no data to `get`):

```
template<typename T, typename ... Rest>
struct DataStructure<T, Rest ...>
{
    ...
    template<size_t idx>
    auto get()
    {
        return GetHelper<idx, DataStructure<T,Rest...>>::get(*this);
    }
    ...
};
```

As you can see this `get` member function is itself templated - this time on the index of the member that is needed (so usage can be things like `data.get<1>()`, similar to `std::tuple`). The actual work is done by a static function in a helper class, `GetHelper`. The reason we can't define the required functionality directly in `DataStructure`'s `get` is because (as we will shortly see) we would need to specialise on `idx` - but it isn't possible to specialise a template member function without specialising the containing class template. Note also the use of a C++14-style `auto` here makes our lives significantly simpler as otherwise we would need quite a complicated expression for the return type.

So on to the helper class. This time we will need an empty forward declaration and two specialisations. First the declaration:

```
template<size_t idx, typename T>
struct GetHelper;
```

Now the base-case (when `idx==0`). In this case we just return the `first` member:

```
template<typename T, typename ... Rest>
struct GetHelper<0, DataStructure<T, Rest ... >>
{
    static T get(DataStructure<T, Rest...>& data)
    {
        return data.first;
    }
};
```

In the recursive case, we decrement `idx` and invoke the `GetHelper` for the `rest` member:

```
template<size_t idx, typename T, typename ... Rest>
struct GetHelper<idx, DataStructure<T, Rest ... >>
{
    static auto get(DataStructure<T, Rest...>& data)
    {
        return GetHelper<idx-1, DataStructure<Rest ...>>::get(data.rest);
    }
};
```

To work through an example, suppose we have `DataStructure<int, float> data` and we need `data.get<1>()`. This invokes `GetHelper<1, DataStructure<int, float>>::get(data)` (the 2nd specialisation), which in turn invokes `GetHelper<0, DataStructure<float>>::get(data.rest)`, which finally returns (by the 1st specialisation as now `idx` is 0) `data.rest.first`.

So that's it! Here is the whole functioning code, with some example use in the `main` function:

```
#include <iostream>

template<size_t idx, typename T>
struct GetHelper;

template<typename ... T>
struct DataStructure
{
};

template<typename T, typename ... Rest>
struct DataStructure<T, Rest ...>
{
    DataStructure(const T& first, const Rest& ... rest)
        : first(first)
        , rest(rest...)
    {}

    T first;
    DataStructure<Rest ... > rest;

    template<size_t idx>
    auto get()
    {
        return GetHelper<idx, DataStructure<T,Rest...>>::get(*this);
    }
};

template<typename T, typename ... Rest>
struct GetHelper<0, DataStructure<T, Rest ... >>
{
    static T get(DataStructure<T, Rest...>& data)
    {
        return data.first;
    }
};

template<size_t idx, typename T, typename ... Rest>
struct GetHelper<idx, DataStructure<T, Rest ... >>
{
    static auto get(DataStructure<T, Rest...>& data)
    {
        return GetHelper<idx-1, DataStructure<Rest ...>>::get(data.rest);
    }
};

int main()
{
    DataStructure<int, float, std::string> data(1, 2.1, "Hello");

    std::cout << data.get<0>() << std::endl;
    std::cout << data.get<1>() << std::endl;
    std::cout << data.get<2>() << std::endl;

    return 0;
}
```

Section 77.4: Argument forwarding

Template may accept both lvalue and rvalue references using *forwarding reference*:

```
template <typename T>
void f(T &&t);
```

In this case, the real type of `t` will be deduced depending on the context:

```
struct X { };

X x;
f(x); // calls f<X&>(x)
f(X()); // calls f<X>(x)
```

In the first case, the type `T` is deduced as *reference to X* (`X&`), and the type of `t` is *lvalue reference to X*, while in the second case the type of `T` is deduced as `X` and the type of `t` as *rvalue reference to X* (`X&&`).

Note: It is worth noticing that in the first case, `decltype(t)` is the same as `T`, but not in the second.

In order to perfectly forward `t` to another function ,whether it is an lvalue or rvalue reference, one must use `std::forward`:

```
template <typename T>
void f(T &&t) {
    g(std::forward<T>(t));
}
```

Forwarding references may be used with variadic templates:

```
template <typename... Args>
void f(Args&&... args) {
    g(std::forward<Args>(args)...);
}
```

Note: Forwarding references can only be used for template parameters, for instance, in the following code, `v` is a rvalue reference, not a forwarding reference:

```
#include <vector>

template <typename T>
void f(std::vector<T> &&v);
```

Section 77.5: Partial template specialization

In contrast of a full template specialization partial template specialization allows to introduce template with some of the arguments of existing template fixed. Partial template specialization is only available for template class/structs:

```
// Common case:
template<typename T, typename U>
struct S {
    T t_val;
    U u_val;
};

// Special case when the first template argument is fixed to int
template<typename V>
struct S<int, V> {
    double another_value;
    int foo(double arg) {// Do something}
```

```
};
```

As shown above, partial template specializations may introduce completely different sets of data and function members.

When a partially specialized template is instantiated, the most suitable specialization is selected. For example, let's define a template and two partial specializations:

```
template<typename T, typename U, typename V>
struct S {
    static void foo() {
        std::cout << "General case\n";
    }
};

template<typename U, typename V>
struct S<int, U, V> {
    static void foo() {
        std::cout << "T = int\n";
    }
};

template<typename V>
struct S<int, double, V> {
    static void foo() {
        std::cout << "T = int, U = double\n";
    }
};
```

Now the following calls:

```
S<std::string, int, double>::foo();
S<int, float, std::string>::foo();
S<int, double, std::string>::foo();
```

will print

```
General case
T = int
T = int, U = double
```

Function templates may only be fully specialized:

```
template<typename T, typename U>
void foo(T t, U u) {
    std::cout << "General case: " << t << " " << u << std::endl;
}

// OK.
template<>
void foo<int, int>(int a1, int a2) {
    std::cout << "Two ints: " << a1 << " " << a2 << std::endl;
}

void invoke_foo() {
    foo(1, 2.1); // Prints "General case: 1 2.1"
    foo(1,2);    // Prints "Two ints: 1 2"
}
```

```
// Compilation error: partial function specialization is not allowed.
template<typename U>
void foo<std::string, U>(std::string t, U u) {
    std::cout << "General case: " << t << " " << u << std::endl;
}
```

Section 77.6: Template Specialization

You can define implementation for specific instantiations of a template class/method.

For example if you have:

```
template <typename T>
T sqrt(T t) { /* Some generic implementation */ }
```

You can then write:

```
template<>
int sqrt<int>(int i) { /* Highly optimized integer implementation */ }
```

Then a user that writes `sqrt(4.0)` will get the generic implementation whereas `sqrt(4)` will get the specialized implementation.

Section 77.7: Alias template

Version ≥ C++11

Basic example:

```
template<typename T> using pointer = T*;
```

This definition makes `pointer<T>` an alias of `T*`. For example:

```
pointer<int> p = new int; // equivalent to: int* p = new int;
```

Alias templates cannot be specialized. However, that functionality can be obtained indirectly by having them refer to a nested type in a struct:

```
template<typename T>
 struct nonconst_pointer_helper { typedef T* type; };

template<typename T>
 struct nonconst_pointer_helper<T const> { typedef T* type; };

template<typename T> using nonconst_pointer = nonconst_pointer_helper<T>::type;
```

Section 77.8: Explicit instantiation

An explicit instantiation definition creates and declares a concrete class, function, or variable from a template, without using it just yet. An explicit instantiation can be referenced from other translation units. This can be used to avoid defining a template in a header file, if it will only be instantiated with a finite set of arguments. For example:

```
// print_string.h
template <class T>
void print_string(const T* str);
```

```
// print_string.cpp
#include "print_string.h"
template void print_string(const char*);
template void print_string(const wchar_t*);
```

Because `print_string<char>` and `print_string<wchar_t>` are explicitly instantiated in `print_string.cpp`, the linker will be able to find them even though the `print_string` template is not defined in the header. If these explicit instantiation declarations were not present, a linker error would likely occur. See Why can templates only be implemented in the header file?

Version ≥ C++11

If an explicit instantiation definition is preceded by the `extern` keyword, it becomes an explicit instantiation *declaration* instead. The presence of an explicit instantiation declaration for a given specialization prevents the implicit instantiation of the given specialization within the current translation unit. Instead, a reference to that specialization that would otherwise cause an implicit instantiation can refer to an explicit instantiation definition in the same or another TU.

`foo.h`

```
#ifndef FOO_H
#define FOO_H
template <class T> void foo(T x) {
    // complicated implementation
}
#endif
```

`foo.cpp`

```
#include "foo.h"
// explicit instantiation definitions for common cases
template void foo(int);
template void foo(double);
```

`main.cpp`

```
#include "foo.h"
// we already know foo.cpp has explicit instantiation definitions for these
extern template void foo(double);
int main() {
    foo(42);   // instantiates foo<int> here;
               // wasteful since foo.cpp provides an explicit instantiation already!
    foo(3.14); // does not instantiate foo<double> here;
               // uses instantiation of foo<double> in foo.cpp instead
}
```

Section 77.9: Non-type template parameter

Apart from types as a template parameter we are allowed to declare values of constant expressions meeting one of the following criteria:

- integral or enumeration type,
- pointer to object or pointer to function,
- lvalue reference to object or lvalue reference to function,
- pointer to member,
- `std::nullptr_t`.

Like all template parameters, non-type template parameters can be explicitly specified, defaulted, or derived implicitly via Template Argument Deduction.

Example of non-type template parameter usage:

```
#include <iostream>

template<typename T, std::size_t size>
std::size_t size_of(T (&anArray)[size])  // Pass array by reference. Requires.
{                                        // an exact size. We allow all sizes
    return size;                         // by using a template "size".
}

int main()
{
    char anArrayOfChar[15];
    std::cout << "anArrayOfChar: " << size_of(anArrayOfChar) << "\n";

    int  anArrayOfData[] = {1,2,3,4,5,6,7,8,9};
    std::cout << "anArrayOfData: " << size_of(anArrayOfData) << "\n";
}
```

Example of explicitly specifying both type and non-type template parameters:

```
#include <array>
int main ()
{
    std::array<int, 5> foo; // int is a type parameter, 5 is non-type
}
```

Non-type template parameters are one of the ways to achieve template recurrence and enables to do Metaprogramming.

Section 77.10: Declaring non-type template arguments with auto

Prior to C++17, when writing a template non-type parameter, you had to specify its type first. So a common pattern became writing something like:

```
template <class T, T N>
struct integral_constant {
    using type = T;
    static constexpr T value = N;
};

using five = integral_constant<int, 5>;
```

But for complicated expressions, using something like this involves having to write `decltype(expr), expr` when instantiating templates. The solution is to simplify this idiom and simply allow `auto`:

Version ≥ C++17

```
template <auto N>
struct integral_constant {
    using type = decltype(N);
    static constexpr type value = N;
};
```

```
using five = integral_constant<5>;
```

Empty custom deleter for unique_ptr

A nice motivating example can come from trying to combine the empty base optimization with a custom deleter for `unique_ptr`. Different C API deleters have different return types, but we don't care - we just want something to work for any function:

```
template <auto DeleteFn>
struct FunctionDeleter {
    template <class T>
    void operator()(T* ptr) const {
        DeleteFn(ptr);
    }
};

template <T, auto DeleteFn>
using unique_ptr_deleter = std::unique_ptr<T, FunctionDeleter<DeleteFn>>;
```

And now you can simply use any function pointer that can take an argument of type `T` as a template non-type parameter, regardless of return type, and get a no-size overhead `unique_ptr` out of it:

```
unique_ptr_deleter<std::FILE, std::fclose> p;
```

Section 77.11: Template template parameters

Sometimes we would like to pass into the template a template type without fixing its values. This is what template template parameters are created for. Very simple template template parameter examples:

```
template <class T>
struct Tag1 { };

template <class T>
struct Tag2 { };

template <template <class> class Tag>
struct IntTag {
   typedef Tag<int> type;
};

int main() {
   IntTag<Tag1>::type t;
}
```

Version ≥ C++11

```
#include <vector>
#include <iostream>

template <class T, template <class...> class C, class U>
C<T> cast_all(const C<U> &c) {
   C<T> result(c.begin(), c.end());
   return result;
}

int main() {
   std::vector<float> vf = {1.2, 2.6, 3.7};
   auto vi = cast_all<int>(vf);
   for(auto &&i: vi) {
      std::cout << i << std::endl;
   }
```

```
}
```

Section 77.12: Default template parameter value

Just like in case of the function arguments, template parameters can have their default values. All template parameters with a default value have to be declared at the end of the template parameter list. The basic idea is that the template parameters with default value can be omitted while template instantiation.

Simple example of default template parameter value usage:

```
template <class T, size_t N = 10>
struct my_array {
    T arr[N];
};

int main() {
    /* Default parameter is ignored, N = 5 */
    my_array<int, 5> a;

    /* Print the length of a.arr: 5 */
    std::cout << sizeof(a.arr) / sizeof(int) << std::endl;

    /* Last parameter is omitted, N = 10 */
    my_array<int> b;

    /* Print the length of a.arr: 10 */
    std::cout << sizeof(b.arr) / sizeof(int) << std::endl;
}
```

Chapter 78: Expression templates

Section 78.1: A basic example illustrating expression templates

An expression template is a compile-time optimization technique used mostly in scientific computing. It's main purpose is to avoid unnecessary temporaries and optimize loop calculations using a single pass (typically when performing operations on numerical aggregates). Expression templates were initially devised in order to circumvent the inefficiencies of naïve operator overloading when implementing numerical `Array` or `Matrix` types. An equivalent terminology for expression templates has been introduced by Bjarne Stroustrup, who calls them "fused operations" in the latest version of his book, "The C++ Programming Language".

Before actually diving into expression templates, you should understand why you need them in the first place. To illustrate this, consider the very simple Matrix class given below:

```
template <typename T, std::size_t COL, std::size_t ROW>
class Matrix {
public:
    using value_type = T;

    Matrix() : values(COL * ROW) {}

    static size_t cols() { return COL; }
    static size_t rows() { return ROW; }

    const T& operator()(size_t x, size_t y) const { return values[y * COL + x]; }
    T& operator()(size_t x, size_t y) { return values[y * COL + x]; }

private:
    std::vector<T> values;
};

template <typename T, std::size_t COL, std::size_t ROW>
Matrix<T, COL, ROW>
operator+(const Matrix<T, COL, ROW>& lhs, const Matrix<T, COL, ROW>& rhs)
{
    Matrix<T, COL, ROW> result;

    for (size_t y = 0; y != lhs.rows(); ++y) {
        for (size_t x = 0; x != lhs.cols(); ++x) {
            result(x, y) = lhs(x, y) + rhs(x, y);
        }
    }
    return result;
}
```

Given the previous class definition, you can now write Matrix expressions such as:

```
const std::size_t cols = 2000;
const std::size_t rows = 1000;

Matrix<double, cols, rows> a, b, c;

// initialize a, b & c
for (std::size_t y = 0; y != rows; ++y) {
    for (std::size_t x = 0; x != cols; ++x) {
        a(x, y) = 1.0;
        b(x, y) = 2.0;
```

```
        c(x, y) = 3.0;
    }
}

Matrix<double, cols, rows> d = a + b + c;  // d(x, y) = 6
```

As illustrated above, being able to overload `operator+()` provides you with a notation which mimics the natural mathematical notation for matrices.

Unfortunately, the previous implementation is also highly inefficient compared to an equivalent "hand-crafted" version.

To understand why, you have to consider what happens when you write an expression such as `Matrix d = a + b + c`. This in fact expands to `((a + b) + c)` or `operator+(operator+(a, b), c)`. In other words, the loop inside `operator+()` is executed twice, whereas it could have been easily performed in a single pass. This also results in 2 temporaries being created, which further degrades performance. In essence, by adding the flexibility to use a notation close to its mathematical counterpart, you have also made the `Matrix` class highly inefficient.

For example, without operator overloading, you could implement a far more efficient Matrix summation using a single pass:

```
template<typename T, std::size_t COL, std::size_t ROW>
Matrix<T, COL, ROW> add3(const Matrix<T, COL, ROW>& a,
                         const Matrix<T, COL, ROW>& b,
                         const Matrix<T, COL, ROW>& c)
{
    Matrix<T, COL, ROW> result;
    for (size_t y = 0; y != ROW; ++y) {
        for (size_t x = 0; x != COL; ++x) {
            result(x, y) = a(x, y) + b(x, y) + c(x, y);
        }
    }
    return result;
}
```

The previous example however has its own disadvantages because it creates a far more convoluted interface for the Matrix class (you would have to consider methods such as `Matrix::add2()`, `Matrix::AddMultiply()` and so on).

Instead let us take a step back and see how we can adapt operator overloading to perform in a more efficient way

The problem stems from the fact that the expression `Matrix d = a + b + c` is evaluated too "eagerly" before you have had an opportunity to build the entire expression tree. In other words, what you really want to achieve is to evaluate `a + b + c` in one pass and only once you actually need to assign the resulting expressing to `d`.

This is the core idea behind expression templates: instead of having `operator+()` evaluate immediately the result of adding two Matrix instances, it will return an *"expression template"* for future evaluation once the entire expression tree has been built.

For example, here is a possible implementation for an expression template corresponding to the summation of 2 types:

```
template <typename LHS, typename RHS>
class MatrixSum
{
public:
    using value_type = typename LHS::value_type;
```

```
    MatrixSum(const LHS& lhs, const RHS& rhs) : rhs(rhs), lhs(lhs) {}

    value_type operator() (int x, int y) const  {
        return lhs(x, y) + rhs(x, y);
    }
private:
    const LHS& lhs;
    const RHS& rhs;
};
```

And here is the updated version of `operator+()`

```
template <typename LHS, typename RHS>
MatrixSum<LHS, RHS> operator+(const LHS& lhs, const LHS& rhs) {
    return MatrixSum<LHS, RHS>(lhs, rhs);
}
```

As you can see, `operator+()` no longer returns an "eager evaluation" of the result of adding 2 Matrix instances (which would be another Matrix instance), but instead an expression template representing the addition operation. The most important point to keep in mind is that the expression has not been evaluated yet. It merely holds references to its operands.

In fact, nothing stops you from instantiating the `MatrixSum<>` expression template as follows:

```
MatrixSum<Matrix<double>, Matrix<double> > SumAB(a, b);
```

You can however at a later stage, when you actually need the result of the summation, evaluate the expression `d = a + b` as follows:

```
for (std::size_t y = 0; y != a.rows(); ++y) {
    for (std::size_t x = 0; x != a.cols(); ++x) {
        d(x, y) = SumAB(x, y);
    }
}
```

As you can see, another benefit of using an expression template, is that you have basically managed to evaluate the sum of `a` and `b` and assign it to `d` in a single pass.

Also, nothing stops you from combining multiple expression templates. For example, `a + b + c` would result in the following expression template:

```
MatrixSum<MatrixSum<Matrix<double>, Matrix<double> >, Matrix<double> > SumABC(SumAB, c);
```

And here again you can evaluate the final result using a single pass:

```
for (std::size_t y = 0; y != a.rows(); ++y) {
    for (std::size_t x = 0; x != a.cols(); ++x) {
        d(x, y) = SumABC(x, y);
    }
}
```

Finally, the last piece of the puzzle is to actually plug your expression template into the `Matrix` class. This is essentially achieved by providing an implementation for `Matrix::operator=()`, which takes the expression template as an argument and evaluates it in one pass, as you did "manually" before:

```
template <typename T, std::size_t COL, std::size_t ROW>
```

```
class Matrix {
public:
    using value_type = T;

    Matrix() : values(COL * ROW) {}

    static size_t cols() { return COL; }
    static size_t rows() { return ROW; }

    const T& operator()(size_t x, size_t y) const { return values[y * COL + x]; }
    T& operator()(size_t x, size_t y) { return values[y * COL + x]; }

    template <typename E>
    Matrix<T, COL, ROW>& operator=(const E& expression) {
        for (std::size_t y = 0; y != rows(); ++y) {
            for (std::size_t x = 0; x != cols(); ++x) {
                values[y * COL + x] = expression(x, y);
            }
        }
        return *this;
    }

private:
    std::vector<T> values;
};
```

Chapter 79: Curiously Recurring Template Pattern (CRTP)

A pattern in which a class inherits from a class template with itself as one of its template parameters. CRTP is usually used to provide *static polymorphism* in C++.

Section 79.1: The Curiously Recurring Template Pattern (CRTP)

CRTP is a powerful, static alternative to virtual functions and traditional inheritance that can be used to give types properties at compile time. It works by having a base class template which takes, as one of its template parameters, the derived class. This permits it to legally perform a `static_cast` of its `this` pointer to the derived class.

Of course, this also means that a CRTP class must *always* be used as the base class of some other class. And the derived class must pass itself to the base class.

Version ≥ C++14

Let's say you have a set of containers that all support the functions `begin()` and `end()`. The standard library's requirements for containers require more functionality. We can design a CRTP base class that provides that functionality, based solely on `begin()` and `end()`:

```
#include <iterator>
template <typename Sub>
class Container {
  private:
    // self() yields a reference to the derived type
    Sub& self() { return *static_cast<Sub*>(this); }
    Sub const& self() const { return *static_cast<Sub const*>(this); }

  public:
    decltype(auto) front() {
      return *self().begin();
    }

    decltype(auto) back() {
      return *std::prev(self().end());
    }

    decltype(auto) size() const {
      return std::distance(self().begin(), self().end());
    }

    decltype(auto) operator[](std::size_t i) {
      return *std::next(self().begin(), i);
    }
};
```

The above class provides the functions `front()`, `back()`, `size()`, and `operator[]` for any subclass which provides `begin()` and `end()`. An example subclass is a simple dynamically allocated array:

```
#include <memory>
// A dynamically allocated array
template <typename T>
class DynArray : public Container<DynArray<T>> {
  public:
```

```
    using Base = Container<DynArray<T>>;

    DynArray(std::size_t size)
      : size_{size},
      data_{std::make_unique<T[]>(size_)}
    { }

    T* begin() { return data_.get(); }
    const T* begin() const { return data_.get(); }
    T* end() { return data_.get() + size_; }
    const T* end() const { return data_.get() + size_; }

  private:
    std::size_t size_;
    std::unique_ptr<T[]> data_;
};
```

Users of the `DynArray` class can use the interfaces provided by the CRTP base class easily as follows:

```
DynArray<int> arr(10);
arr.front() = 2;
arr[2] = 5;
assert(arr.size() == 10);
```

Usefulness: This pattern particularly avoids virtual function calls at run-time which occur to traverse down the inheritance hierarchy and simply relies on static casts:

```
DynArray<int> arr(10);
DynArray<int>::Base & base = arr;
base.begin(); // no virtual calls
```

The only static cast inside the function `begin()` in the base class `Container<DynArray<int>>` allows the compiler to drastically optimize the code and no virtual table look up happens at runtime.

Limitations: Because the base class is templated and different for two different `DynArray`s it is not possible to store pointers to their base classes in an type-homogenous array as one could generally do with normal inheritance where the base class is not dependent on the derived type:

```
class A {};
class B: public A{};

A* a = new B;
```

Section 79.2: CRTP to avoid code duplication

The example in Visitor Pattern provides a compelling use-case for CRTP:

```
struct IShape
{
    virtual ~IShape() = default;

    virtual void accept(IShapeVisitor&) const = 0;
};

struct Circle : IShape
{
    // ...
    // Each shape has to implement this method the same way
```

```
    void accept(IShapeVisitor& visitor) const override { visitor.visit(*this); }
    // ...
};

struct Square : IShape
{
    // ...
    // Each shape has to implement this method the same way
    void accept(IShapeVisitor& visitor) const override { visitor.visit(*this); }
    // ...
};
```

Each child type of `IShape` needs to implement the same function the same way. That's a lot of extra typing. Instead, we can introduce a new type in the hierarchy that does this for us:

```
template <class Derived>
struct IShapeAcceptor : IShape {
    void accept(IShapeVisitor& visitor) const override {
        // visit with our exact type
        visitor.visit(*static_cast<Derived const*>(this));
    }
};
```

And now, each shape simply needs to inherit from the acceptor:

```
struct Circle : IShapeAcceptor<Circle>
{
    Circle(const Point& center, double radius) : center(center), radius(radius) {}
    Point center;
    double radius;
};

struct Square : IShapeAcceptor<Square>
{
    Square(const Point& topLeft, double sideLength) : topLeft(topLeft), sideLength(sideLength) {}

    Point topLeft;
    double sideLength;
};
```

No duplicate code necessary.

Chapter 80: Threading

Parameter	Details
`other`	Takes ownership of `other`, `other` doesn't own the thread anymore
`func`	Function to call in a separate thread
`args`	Arguments for `func`

Section 80.1: Creating a std::thread

In C++, threads are created using the std::thread class. A thread is a separate flow of execution; it is analogous to having a helper perform one task while you simultaneously perform another. When all the code in the thread is executed, it *terminates*. When creating a thread, you need to pass something to be executed on it. A few things that you can pass to a thread:

- Free functions
- Member functions
- Functor objects
- Lambda expressions

Free function example - executes a function on a separate thread (Live Example):

```
#include <iostream>
#include <thread>

void foo(int a)
{
    std::cout << a << '\n';
}

int main()
{
    // Create and execute the thread
    std::thread thread(foo, 10); // foo is the function to execute, 10 is the
                                 // argument to pass to it

    // Keep going; the thread is executed separately

    // Wait for the thread to finish; we stay here until it is done
    thread.join();

    return 0;
}
```

Member function example - executes a member function on a separate thread (Live Example):

```
#include <iostream>
#include <thread>

class Bar
{
public:
    void foo(int a)
    {
        std::cout << a << '\n';
    }
};
```

```
int main()
{
    Bar bar;

    // Create and execute the thread
    std::thread thread(&Bar::foo, &bar, 10); // Pass 10 to member function

    // The member function will be executed in a separate thread

    // Wait for the thread to finish, this is a blocking operation
    thread.join();

    return 0;
}
```

Functor object example (Live Example):

```
#include <iostream>
#include <thread>

class Bar
{
public:
    void operator()(int a)
    {
        std::cout << a << '\n';
    }
};

int main()
{
    Bar bar;

    // Create and execute the thread
    std::thread thread(bar, 10); // Pass 10 to functor object

    // The functor object will be executed in a separate thread

    // Wait for the thread to finish, this is a blocking operation
    thread.join();

    return 0;
}
```

Lambda expression example (Live Example):

```
#include <iostream>
#include <thread>

int main()
{
    auto lambda = [](int a) { std::cout << a << '\n'; };

    // Create and execute the thread
    std::thread thread(lambda, 10); // Pass 10 to the lambda expression

    // The lambda expression will be executed in a separate thread

    // Wait for the thread to finish, this is a blocking operation
    thread.join();
```

```
    return 0;
}
```

Section 80.2: Passing a reference to a thread

You cannot pass a reference (or `const` reference) directly to a thread because `std::thread` will copy/move them. Instead, use `std::reference_wrapper`:

```
void foo(int& b)
{
    b = 10;
}

int a = 1;
std::thread thread{ foo, std::ref(a) }; //'a' is now really passed as reference

thread.join();
std::cout << a << '\n'; //Outputs 10
```

```
void bar(const ComplexObject& co)
{
    co.doCalculations();
}

ComplexObject object;
std::thread thread{ bar, std::cref(object) }; //'object' is passed as const&

thread.join();
std::cout << object.getResult() << '\n'; //Outputs the result
```

Section 80.3: Using std::async instead of std::thread

`std::async` is also able to make threads. Compared to `std::thread` it is considered less powerful but easier to use when you just want to run a function asynchronously.

Asynchronously calling a function

```
#include <future>
#include <iostream>

unsigned int square(unsigned int i){
    return i*i;
}

int main() {
    auto f = std::async(std::launch::async, square, 8);
    std::cout << "square currently running\n"; //do something while square is running
    std::cout << "result is " << f.get() << '\n'; //getting the result from square
}
```

Common Pitfalls

- `std::async` returns a `std::future` that holds the return value that will be calculated by the function. When that `future` gets destroyed it waits until the thread completes, making your code effectively single threaded. This is easily overlooked when you don't need the return value:

```
std::async(std::launch::async, square, 5);
//thread already completed at this point, because the returning future got destroyed
```

- `std::async` works without a launch policy, so `std::async(square, 5);` compiles. When you do that the system gets to decide if it wants to create a thread or not. The idea was that the system chooses to make a thread unless it is already running more threads than it can run efficiently. Unfortunately implementations commonly just choose not to create a thread in that situation, ever, so you need to override that behavior with `std::launch::async` which forces the system to create a thread.
- Beware of race conditions.

More on async on Futures and Promises

Section 80.4: Basic Synchronization

Thread synchronization can be accomplished using mutexes, among other synchronization primitives. There are several mutex types provided by the standard library, but the simplest is `std::mutex`. To lock a mutex, you construct a lock on it. The simplest lock type is `std::lock_guard`:

```
std::mutex m;
void worker() {
    std::lock_guard<std::mutex> guard(m); // Acquires a lock on the mutex
    // Synchronized code here
} // the mutex is automatically released when guard goes out of scope
```

With `std::lock_guard` the mutex is locked for the whole lifetime of the lock object. In cases where you need to manually control the regions for locking, use `std::unique_lock` instead:

```
std::mutex m;
void worker() {
    // by default, constructing a unique_lock from a mutex will lock the mutex
    // by passing the std::defer_lock as a second argument, we
    // can construct the guard in an unlocked state instead and
    // manually lock later.
    std::unique_lock<std::mutex> guard(m, std::defer_lock);
    // the mutex is not locked yet!
    guard.lock();
    // critical section
    guard.unlock();
    // mutex is again released
}
```

More Thread synchronization structures

Section 80.5: Create a simple thread pool

C++11 threading primitives are still relatively low level. They can be used to write a higher level construct, like a thread pool:

Version ≥ C++14

```
struct tasks {
  // the mutex, condition variable and deque form a single
  // thread-safe triggered queue of tasks:
  std::mutex m;
  std::condition_variable v;
  // note that a packaged_task<void> can store a packaged_task<R>:
```

```
  std::deque<std::packaged_task<void()>> work;

  // this holds futures representing the worker threads being done:
  std::vector<std::future<void>> finished;

  // queue( lambda ) will enqueue the lambda into the tasks for the threads
  // to use.  A future of the type the lambda returns is given to let you get
  // the result out.
  template<class F, class R=std::result_of_t<F&()>>
  std::future<R> queue(F&& f) {
    // wrap the function object into a packaged task, splitting
    // execution from the return value:
    std::packaged_task<R()> p(std::forward<F>(f));

    auto r=p.get_future(); // get the return value before we hand off the task
    {
      std::unique_lock<std::mutex> l(m);
      work.emplace_back(std::move(p)); // store the task<R()> as a task<void()>
    }
    v.notify_one(); // wake a thread to work on the task

    return r; // return the future result of the task
  }

  // start N threads in the thread pool.
  void start(std::size_t N=1){
    for (std::size_t i = 0; i < N; ++i)
    {
      // each thread is a std::async running this->thread_task():
      finished.push_back(
        std::async(
          std::launch::async,
          [this]{ thread_task(); }
        )
      );
    }
  }
  // abort() cancels all non-started tasks, and tells every working thread
  // stop running, and waits for them to finish up.
  void abort() {
    cancel_pending();
    finish();
  }
  // cancel_pending() merely cancels all non-started tasks:
  void cancel_pending() {
    std::unique_lock<std::mutex> l(m);
    work.clear();
  }
  // finish enques a "stop the thread" message for every thread, then waits for them:
  void finish() {
    {
      std::unique_lock<std::mutex> l(m);
      for(auto&&unused:finished){
        work.push_back({});
      }
    }
    v.notify_all();
    finished.clear();
  }
  ~tasks() {
    finish();
  }
```

```
private:
  // the work that a worker thread does:
  void thread_task() {
    while(true){
      // pop a task off the queue:
      std::packaged_task<void()> f;
      {
        // usual thread-safe queue code:
        std::unique_lock<std::mutex> l(m);
        if (work.empty()){
          v.wait(l,[&]{return !work.empty();});
        }
        f = std::move(work.front());
        work.pop_front();
      }
      // if the task is invalid, it means we are asked to abort:
      if (!f.valid()) return;
      // otherwise, run the task:
      f();
    }
  }
};
```

`tasks.queue( []{ return "hello world"s; } )` returns a `std::future<std::string>`, which when the tasks object gets around to running it is populated with `hello world`.

You create threads by running `tasks.start(10)` (which starts 10 threads).

The use of `packaged_task<void()>` is merely because there is no type-erased `std::function` equivalent that stores move-only types. Writing a custom one of those would probably be faster than using `packaged_task<void()>`.

Live example.

Version = C++11

In C++11, replace `result_of_t<blah>` with `typename result_of<blah>::type`.

More on Mutexes.

Section 80.6: Ensuring a thread is always joined

When the destructor for `std::thread` is invoked, a call to either `join()` or `detach()` **must** have been made. If a thread has not been joined or detached, then by default `std::terminate` will be called. Using RAII, this is generally simple enough to accomplish:

```
class thread_joiner
{
public:

    thread_joiner(std::thread t)
        : t_(std::move(t))
    { }

    ~thread_joiner()
    {
        if(t_.joinable()) {
            t_.join();
        }
```

```
    }

private:

    std::thread t_;
}
```

This is then used like so:

```
void perform_work()
{
    // Perform some work
}

void t()
{
    thread_joiner j{std::thread(perform_work)};
    // Do some other calculations while thread is running
} // Thread is automatically joined here
```

This also provides exception safety; if we had created our thread normally and the work done in `t()` performing other calculations had thrown an exception, `join()` would never have been called on our thread and our process would have been terminated.

Section 80.7: Operations on the current thread

`std::this_thread` is a `namespace` which has functions to do interesting things on the current thread from function it is called from.

Function	Description
`get_id`	Returns the id of the thread
`sleep_for`	Sleeps for a specified amount of time
`sleep_until`	Sleeps until a specific time
`yield`	Reschedule running threads, giving other threads priority

Getting the current threads id using `std::this_thread::get_id`:

```
void foo()
{
    //Print this threads id
    std::cout << std::this_thread::get_id() << '\n';
}

std::thread thread{ foo };
thread.join(); //'threads' id has now been printed, should be something like 12556

foo(); //The id of the main thread is printed, should be something like 2420
```

Sleeping for 3 seconds using `std::this_thread::sleep_for`:

```
void foo()
{
    std::this_thread::sleep_for(std::chrono::seconds(3));
}
```

```
std::thread thread{ foo };
foo.join();

std::cout << "Waited for 3 seconds!\n";
```

Sleeping until 3 hours in the future using `std::this_thread::sleep_until`:

```
void foo()
{
    std::this_thread::sleep_until(std::chrono::system_clock::now() + std::chrono::hours(3));
}

std::thread thread{ foo };
thread.join();

std::cout << "We are now located 3 hours after the thread has been called\n";
```

Letting other threads take priority using `std::this_thread::yield`:

```
void foo(int a)
{
    for (int i = 0; i < al ++i)
        std::this_thread::yield(); //Now other threads take priority, because this thread
                                   //isn't doing anything important

    std::cout << "Hello World!\n";
}

std::thread thread{ foo, 10 };
thread.join();
```

Section 80.8: Using Condition Variables

A condition variable is a primitive used in conjunction with a mutex to orchestrate communication between threads. While it is neither the exclusive or most efficient way to accomplish this, it can be among the simplest to those familiar with the pattern.

One waits on a `std::condition_variable` with a `std::unique_lock<std::mutex>`. This allows the code to safely examine shared state before deciding whether or not to proceed with acquisition.

Below is a producer-consumer sketch that uses `std::thread`, `std::condition_variable`, `std::mutex`, and a few others to make things interesting.

```
#include <condition_variable>
#include <cstddef>
#include <iostream>
#include <mutex>
#include <queue>
#include <random>
#include <thread>


int main()
{
    std::condition_variable cond;
    std::mutex mtx;
```

```
    std::queue<int> intq;
    bool stopped = false;

    std::thread producer{[&]()
    {
        // Prepare a random number generator.
        // Our producer will simply push random numbers to intq.
        //
        std::default_random_engine gen{};
        std::uniform_int_distribution<int> dist{};

        std::size_t count = 4006;
        while(count--)
        {
            // Always lock before changing
            // state guarded by a mutex and
            // condition_variable (a.k.a. "condvar").
            std::lock_guard<std::mutex> L{mtx};

            // Push a random int into the queue
            intq.push(dist(gen));

            // Tell the consumer it has an int
            cond.notify_one();
        }

        // All done.
        // Acquire the lock, set the stopped flag,
        // then inform the consumer.
        std::lock_guard<std::mutex> L{mtx};

        std::cout << "Producer is done!" << std::endl;

        stopped = true;
        cond.notify_one();
    }};

    std::thread consumer{[&]()
    {
        do{
            std::unique_lock<std::mutex> L{mtx};
            cond.wait(L,[&]()
            {
                // Acquire the lock only if
                // we've stopped or the queue
                // isn't empty
                return stopped || ! intq.empty();
            });

            // We own the mutex here; pop the queue
            // until it empties out.

            while( ! intq.empty())
            {
                const auto val = intq.front();
                intq.pop();

                std::cout << "Consumer popped: " << val << std::endl;
            }

            if(stopped){
                // producer has signaled a stop
```

```
                std::cout << "Consumer is done!" << std::endl;
                break;
            }

        }while(true);
    }};

    consumer.join();
    producer.join();

    std::cout << "Example Completed!" << std::endl;

    return 0;
}
```

Section 80.9: Thread operations

When you start a thread, it will execute until it is finished.

Often, at some point, you need to (possibly - the thread may already be done) wait for the thread to finish, because you want to use the result for example.

```
int n;
std::thread thread{ calculateSomething, std::ref(n) };

//Doing some other stuff

//We need 'n' now!
//Wait for the thread to finish - if it is not already done
thread.join();

//Now 'n' has the result of the calculation done in the separate thread
std::cout << n << '\n';
```

You can also `detach` the thread, letting it execute freely:

```
std::thread thread{ doSomething };

//Detaching the thread, we don't need it anymore (for whatever reason)
thread.detach();

//The thread will terminate when it is done, or when the main thread returns
```

Section 80.10: Thread-local storage

Thread-local storage can be created using the `thread_local` keyword. A variable declared with the `thread_local` specifier is said to have **thread storage duration.**

- Each thread in a program has its own copy of each thread-local variable.
- A thread-local variable with function (local) scope will be initialized the first time control passes through its definition. Such a variable is implicitly static, unless declared `extern`.
- A thread-local variable with namespace or class (non-local) scope will be initialized as part of thread startup.
- Thread-local variables are destroyed upon thread termination.
- A member of a class can only be thread-local if it is static. There will therefore be one copy of that variable per thread, rather than one copy per (thread, instance) pair.

Example:

```
void debug_counter() {
    thread_local int count = 0;
    Logger::log("This function has been called %d times by this thread", ++count);
}
```

Section 80.11: Reassigning thread objects

We can create empty thread objects and assign work to them later.

If we assign a thread object to another active, `joinable` thread, `std::terminate` will automatically be called before the thread is replaced.

```
#include <thread>

void foo()
{
    std::this_thread::sleep_for(std::chrono::seconds(3));
}
//create 100 thread objects that do nothing
std::thread executors[100];

// Some code

// I want to create some threads now

for (int i = 0;i < 100;i++)
{
    // If this object doesn't have a thread assigned
    if (!executors[i].joinable())
         executors[i] = std::thread(foo);
}
```

Chapter 81: Thread synchronization structures

Working with threads might need some synchronization techniques if the threads interact. In this topic, you can find the different structures which are provided by the standard library to solve these issues.

Section 81.1: std::condition_variable_any, std::cv_status

A generalization of `std::condition_variable`, `std::condition_variable_any` works with any type of BasicLockable structure.

`std::cv_status` as a return status for a condition variable has two possible return codes:

- std::cv_status::no_timeout: There was no timeout, condition variable was notified
- std::cv_status::no_timeout: Condition variable timed out

Section 81.2: std::shared_lock

A shared_lock can be used in conjunction with a unique lock to allow multiple readers and exclusive writers.

```
#include <unordered_map>
#include <mutex>
#include <shared_mutex>
#include <thread>
#include <string>
#include <iostream>

class PhoneBook {
    public:
        string getPhoneNo( const std::string & name )
        {
            shared_lock<shared_timed_mutex> r(_protect);
            auto it =  _phonebook.find( name );
            if ( it == _phonebook.end() )
                return (*it).second;
            return "";
        }
        void addPhoneNo ( const std::string & name, const std::string & phone )
        {
            unique_lock<shared_timed_mutex> w(_protect);
            _phonebook[name] = phone;
        }

        shared_timed_mutex _protect;
        unordered_map<string,string>  _phonebook;
    };
```

Section 81.3: std::call_once, std::once_flag

`std::call_once` ensures execution of a function exactly once by competing threads. It throws `std::system_error` in case it cannot complete its task.

Used in conjunction with `std::once_flag`.

```
#include <mutex>
```

```
#include <iostream>

std::once_flag flag;
void do_something(){
      std::call_once(flag, [](){std::cout << "Happens once" << std::endl;});

      std::cout << "Happens every time" << std::endl;
}
```

Section 81.4: Object locking for efficient access

Often you want to lock the entire object while you perform multiple operations on it. For example, if you need to examine or modify the object using *iterators*. Whenever you need to call multiple member functions it is generally more efficient to lock the whole object rather than individual member functions.

For example:

```
class text_buffer
{
    // for readability/maintainability
    using mutex_type = std::shared_timed_mutex;
    using reading_lock = std::shared_lock<mutex_type>;
    using updates_lock = std::unique_lock<mutex_type>;

public:
    // This returns a scoped lock that can be shared by multiple
    // readers at the same time while excluding any writers
    [[nodiscard]]
    reading_lock lock_for_reading() const { return reading_lock(mtx); }

    // This returns a scoped lock that is exclusing to one
    // writer preventing any readers
    [[nodiscard]]
    updates_lock lock_for_updates() { return updates_lock(mtx); }

    char* data() { return buf; }
    char const* data() const { return buf; }

    char* begin() { return buf; }
    char const* begin() const { return buf; }

    char* end() { return buf + sizeof(buf); }
    char const* end() const { return buf + sizeof(buf); }

    std::size_t size() const { return sizeof(buf); }

private:
    char buf[1024];
    mutable mutex_type mtx; // mutable allows const objects to be locked
};
```

When calculating a checksum the object is locked for reading, allowing other threads that want to read from the object at the same time to do so.

```
std::size_t checksum(text_buffer const& buf)
{
    std::size_t sum = 0xA44944A4;

    // lock the object for reading
```

```
    auto lock = buf.lock_for_reading();

    for(auto c: buf)
        sum = (sum << 8) | (((unsigned char) ((sum & 0xFF000000) >> 24)) ^ c);

    return sum;
}
```

Clearing the object updates its internal data so it must be done using an exclusing lock.

```
void clear(text_buffer& buf)
{
    auto lock = buf.lock_for_updates(); // exclusive lock
    std::fill(std::begin(buf), std::end(buf), '\0');
}
```

When obtaining more than one lock care should be taken to always acquire the locks in the same order for all threads.

```
void transfer(text_buffer const& input, text_buffer& output)
{
    auto lock1 = input.lock_for_reading();
    auto lock2 = output.lock_for_updates();

    std::copy(std::begin(input), std::end(input), std::begin(output));
}
```

note: This is best done using std::deferred::lock and calling std::lock

Chapter 82: The Rule of Three, Five, And Zero

Section 82.1: Rule of Zero

Version ≥ C++11

We can combine the principles of the Rule of Five and RAII to get a much leaner interface: the Rule of Zero: any resource that needs to be managed should be in its own type. That type would have to follow the Rule of Five, but all users of that resource do not need to write *any* of the five special member functions and can simply `default` all of them.

Using the `Person` class introduced in the Rule of Three example, we can create a resource-managing object for `cstrings`:

```
class cstring {
private:
    char* p;

public:
    ~cstring() { delete [] p; }
    cstring(cstring const& );
    cstring(cstring&& );
    cstring& operator=(cstring const& );
    cstring& operator=(cstring&& );

    /* other members as appropriate */
};
```

And once this is separate, our `Person` class becomes far simpler:

```
class Person {
    cstring name;
    int arg;

public:
    ~Person() = default;
    Person(Person const& ) = default;
    Person(Person&& ) = default;
    Person& operator=(Person const& ) = default;
    Person& operator=(Person&& ) = default;

    /* other members as appropriate */
};
```

The special members in `Person` do not even need to be declared explicitly; the compiler will default or delete them appropriately, based on the contents of `Person`. Therefore, the following is also an example of the rule of zero.

```
struct Person {
    cstring name;
    int arg;
};
```

If `cstring` were to be a move-only type, with a `delete`d copy constructor/assignment operator, then `Person` would automatically be move-only as well.

The term rule of zero was introduced by R. Martinho Fernandes

Section 82.2: Rule of Five

Version ≥ C++11

C++11 introduces two new special member functions: the move constructor and the move assignment operator. For all the same reasons that you want to follow the Rule of Three in C++03, you usually want to follow the Rule of Five in C++11: If a class requires ONE of five special member functions, and if move semantics are desired, then it most likely requires ALL FIVE of them.

Note, however, that failing to follow the Rule of Five is usually not considered an error, but a missed optimisation opportunity, as long as the Rule of Three is still followed. If no move constructor or move assignment operator is available when the compiler would normally use one, it will instead use copy semantics if possible, resulting in a less efficient operation due to unnecessary copy operations. If move semantics aren't desired for a class, then it has no need to declare a move constructor or assignment operator.

Same example as for the Rule of Three:

```
class Person
{
    char* name;
    int age;

public:
    // Destructor
    ~Person() { delete [] name; }

    // Implement Copy Semantics
    Person(Person const& other)
        : name(new char[std::strlen(other.name) + 1])
        , age(other.age)
    {
        std::strcpy(name, other.name);
    }

    Person &operator=(Person const& other)
    {
        // Use copy and swap idiom to implement assignment.
        Person copy(other);
        swap(*this, copy);
        return *this;
    }

    // Implement Move Semantics
    // Note: It is usually best to mark move operators as noexcept
    //       This allows certain optimizations in the standard library
    //       when the class is used in a container.

    Person(Person&& that) noexcept
        : name(nullptr)               // Set the state so we know it is undefined
        , age(0)
    {
        swap(*this, that);
    }

    Person& operator=(Person&& that) noexcept
    {
```

```
        swap(*this, that);
        return *this;
    }

    friend void swap(Person& lhs, Person& rhs) noexcept
    {
        std::swap(lhs.name, rhs.name);
        std::swap(lhs.age, rhs.age);
    }
};
```

Alternatively, both the copy and move assignment operator can be replaced with a single assignment operator, which takes an instance by value instead of reference or rvalue reference to facilitate using the copy-and-swap idiom.

```
Person& operator=(Person copy)
{
    swap(*this, copy);
    return *this;
}
```

Extending from the Rule of Three to the Rule of Five is important for performance reasons, but is not strictly necessary in most cases. Adding the copy constructor and assignment operator ensures that moving the type will not leak memory (move-constructing will simply fall back to copying in that case), but will be performing copies that the caller probably did not anticipate.

Section 82.3: Rule of Three

Version ≤ c++03

The Rule of Three states that if a type ever needs to have a user-defined copy constructor, copy assignment operator, or destructor, then it must have *all three*.

The reason for the rule is that a class which needs any of the three manages some resource (file handles, dynamically allocated memory, etc), and all three are needed to manage that resource consistently. The copy functions deal with how the resource gets copied between objects, and the destructor would destroy the resource, in accord with RAII principles.

Consider a type that manages a string resource:

```
class Person
{
    char* name;
    int age;

public:
    Person(char const* new_name, int new_age)
        : name(new char[std::strlen(new_name) + 1])
        , age(new_age)
    {
       std::strcpy(name, new_name);
    }

    ~Person() {
        delete [] name;
    }
};
```

Since `name` was allocated in the constructor, the destructor deallocates it to avoid leaking memory. But what happens if such an object is copied?

```
int main()
{
    Person p1("foo", 11);
    Person p2 = p1;
}
```

First, `p1` will be constructed. Then `p2` will be copied from `p1`. However, the C++-generated copy constructor will copy each component of the type as-is. Which means that `p1.name` and `p2.name` both point to the **same** string.

When `main` ends, destructors will be called. First `p2`'s destructor will be called; it will delete the string. Then `p1`'s destructor will be called. However, the string is *already deleted*. Calling `delete` on memory that was already deleted yields undefined behavior.

To avoid this, it is necessary to provide a suitable copy constructor. One approach is to implement a reference counted system, where different `Person` instances share the same string data. Each time a copy is performed, the shared reference count is incremented. The destructor then decrements the reference count, only releasing the memory if the count is zero.

Or we could implement value semantics and deep copying behavior:

```
Person(Person const& other)
    : name(new char[std::strlen(other.name) + 1])
    , age(other.age)
{
    std::strcpy(name, other.name);
}

Person &operator=(Person const& other)
{
    // Use copy and swap idiom to implement assignment
    Person copy(other);
    swap(copy);            //  assume swap() exchanges contents of *this and copy
    return *this;
}
```

Implementation of the copy assignment operator is complicated by the need to release an existing buffer. The copy and swap technique creates a temporary object which holds a new buffer. Swapping the contents of `*this` and `copy` gives ownership to `copy` of the original buffer. Destruction of `copy`, as the function returns, releases the buffer previously owned by `*this`.

Section 82.4: Self-assignment Protection

When writing a copy assignment operator, it is *very* important that it be able to work in the event of self-assignment. That is, it has to allow this:

```
SomeType t = ...;
t = t;
```

Self-assignment usually doesn't happen in such an obvious way. It typically happens via a circuitous route through various code systems, where the location of the assignment simply has two `Person` pointers or references and has no idea that they are the same object.

Any copy assignment operator you write must be able to take this into account.

The typical way to do so is to wrap all of the assignment logic in a condition like this:

```
SomeType &operator=(const SomeType &other)
{
    if(this != &other)
    {
        //Do assignment logic.
    }
    return *this;
}
```

Note: It is important to think about self-assignment and ensure that your code behaves correctly when it happens. However, self-assignment is a very rare occurrence and optimizing to prevent it may actually pessimize the normal case. Since the normal case is much more common, pessimizing for self-assignment may well reduce your code efficiency (so be careful using it).

As an example, the normal technique for implementing the assignment operator is the `copy and swap idiom`. The normal implementation of this technique does not bother to test for self-assignment (even though self-assignment is expensive because a copy is made). The reason is that pessimization of the normal case has been shown to be much more costly (as it happens more often).

Version ≥ c++11

Move assignment operators must also be protected against self-assignment. However, the logic for many such operators is based on `std::swap`, which can handle swapping from/to the same memory just fine. So if your move assignment logic is nothing more than a series of swap operations, then you do not need self-assignment protection.

If this is not the case, you *must* take similar measures as above.

Chapter 83: RAII: Resource Acquisition Is Initialization

Section 83.1: Locking

Bad locking:

```
std::mutex mtx;

void bad_lock_example() {
    mtx.lock();
    try
    {
        foo();
        bar();
        if (baz()) {
            mtx.unlock();   // Have to unlock on each exit point.
            return;
        }
        quux();
        mtx.unlock();       // Normal unlock happens here.
    }
    catch(...) {
        mtx.unlock();       // Must also force unlock in the presence of
        throw;              // exceptions and allow the exception to continue.
    }
}
```

That is the wrong way to implement the locking and unlocking of the mutex. To ensure the correct release of the mutex with `unlock()` requires the programer to make sure that all the flows resulting in the exiting of the function result in a call to `unlock()`. As shown above this is a brittle processes as it requires any maintainers to continue following the pattern manually.

Using an appropriately crafted class to implement RAII, the problem is trivial:

```
std::mutex mtx;

void good_lock_example() {
    std::lock_guard<std::mutex> lk(mtx);   // constructor locks.
                                           // destructor unlocks. destructor call
                                           // guaranteed by language.

    foo();
    bar();
    if (baz()) {
        return;
    }
    quux();
}
```

`lock_guard` is an extremely simple class template that simply calls `lock()` on its argument in its constructor, keeps a reference to the argument, and calls `unlock()` on the argument in its destructor. That is, when the `lock_guard` goes out of scope, the `mutex` is guaranteed to be unlocked. It doesn't matter if the reason it went out of scope is an exception or an early return - all cases are handled; regardless of the control flow, we have guaranteed that we will unlock correctly.

Section 83.2: ScopeSuccess (c++17)

Version ≥ C++17

Thanks to `int std::uncaught_exceptions()`, we can implement action which executes only on success (no thrown exception in scope). Previously `bool std::uncaught_exception()` just allows to detect if **any** stack unwinding is running.

```
#include <exception>
#include <iostream>

template <typename F>
class ScopeSuccess
{
private:
    F f;
    int uncaughtExceptionCount = std::uncaught_exceptions();
public:
    explicit ScopeSuccess(const F& f) : f(f) {}
    ScopeSuccess(const ScopeSuccess&) = delete;
    ScopeSuccess& operator =(const ScopeSuccess&) = delete;

    // f() might throw, as it can be caught normally.
    ~ScopeSuccess() noexcept(noexcept(f())) {
        if (uncaughtExceptionCount == std::uncaught_exceptions()) {
            f();
        }
    }
};

struct Foo {
    ~Foo() {
        try {
            ScopeSuccess logSuccess{[](){std::cout << "Success 1\n";}};
            // Scope succeeds,
            // even if Foo is destroyed during stack unwinding
            // (so when 0 < std::uncaught_exceptions())
            // (or previously std::uncaught_exception() == true)
        } catch (...) {
        }
        try {
            ScopeSuccess logSuccess{[](){std::cout << "Success 2\n";}};

            throw std::runtime_error("Failed"); // returned value
                                                // of std::uncaught_exceptions increases
        } catch (...) { // returned value of std::uncaught_exceptions decreases
        }
    }

};

int main()
{
    try {
        Foo foo;

        throw std::runtime_error("Failed"); // std::uncaught_exceptions() == 1
    } catch (...) { // std::uncaught_exceptions() == 0
    }
}
```

Output:

```
Success 1
```

Section 83.3: ScopeFail (c++17)

Version ≥ C++17

Thanks to `int std::uncaught_exceptions()`, we can implement action which executes only on failure (thrown exception in scope). Previously `bool std::uncaught_exception()` just allows to detect if **any** stack unwinding is running.

```
#include <exception>
#include <iostream>

template <typename F>
class ScopeFail
{
private:
    F f;
    int uncaughtExceptionCount = std::uncaught_exceptions();
public:
    explicit ScopeFail(const F& f) : f(f) {}
    ScopeFail(const ScopeFail&) = delete;
    ScopeFail& operator =(const ScopeFail&) = delete;

    // f() should not throw, else std::terminate is called.
    ~ScopeFail() {
        if (uncaughtExceptionCount != std::uncaught_exceptions()) {
            f();
        }
    }
};

struct Foo {
    ~Foo() {
        try {
            ScopeFail logFailure{[](){std::cout << "Fail 1\n";}};
            // Scope succeeds,
            // even if Foo is destroyed during stack unwinding
            // (so when 0 < std::uncaught_exceptions())
            // (or previously std::uncaught_exception() == true)
        } catch (...) {
        }
        try {
            ScopeFail logFailure{[](){std::cout << "Failure 2\n";}};

            throw std::runtime_error("Failed"); // returned value
                                                // of std::uncaught_exceptions increases
        } catch (...) { // returned value of std::uncaught_exceptions decreases
        }
    }

};

int main()
{
    try {
        Foo foo;
```

```
        throw std::runtime_error("Failed"); // std::uncaught_exceptions() == 1
    } catch (...) { // std::uncaught_exceptions() == 0
    }
}
```

Output:

```
Failure 2
```

Section 83.4: Finally/ScopeExit

For cases when we don't want to write special classes to handle some resource, we may write a generic class:

```
template<typename Function>
class Finally final
{
public:
    explicit Finally(Function f) : f(std::move(f)) {}
    ~Finally() { f(); } // (1) See below

    Finally(const Finally&) = delete;
    Finally(Finally&&) = default;
    Finally& operator =(const Finally&) = delete;
    Finally& operator =(Finally&&) = delete;
private:
    Function f;
};
// Execute the function f when the returned object goes out of scope.
template<typename Function>
auto onExit(Function &&f) { return Finally<std::decay_t<Function>>{std::forward<Function>(f)}; }
```

And its example usage

```
void foo(std::vector<int>& v, int i)
{
    // ...

    v[i] += 42;
    auto autoRollBackChange = onExit([&](){ v[i] -= 42; });

    // ... code as recursive call `foo(v, i + 1)`
}
```

Note (1): Some discussion about destructor definition has to be considered to handle exception:

- `~Finally() noexcept { f(); }`: `std::terminate` is called in case of exception
- `~Finally() noexcept(noexcept(f())) { f(); }`: terminate() is called only in case of exception during stack unwinding.
- `~Finally() noexcept { try { f(); } catch (...) { /* ignore exception (might log it) */} }` No `std::terminate` called, but we cannot handle error (even for non stack unwinding).

Chapter 84: RTTI: Run-Time Type Information

Section 84.1: dynamic_cast

Use `dynamic_cast<>()` as a function, which helps you to cast down through an inheritance hierarchy (main description).

If you must do some non-polymorphic work on some derived classes `B` and `C`, but received the base `class A`, then write like this:

```
class A { public: virtual ~A(){} };

class B: public A
{ public: void work4B(){} };

class C: public A
{ public: void work4C(){} };

void non_polymorphic_work(A* ap)
{
  if (B* bp =dynamic_cast<B*>(ap))
    bp->work4B();
  if (C* cp =dynamic_cast<C*>(ap))
    cp->work4C();
}
```

Section 84.2: The typeid keyword

The `typeid` keyword is a unary operator that yields run-time type information about its operand if the operand's type is a polymorphic class type. It returns an lvalue of type `const std::type_info`. Top-level cv-qualification are ignored.

```
struct Base {
    virtual ~Base() = default;
};
struct Derived : Base {};
Base* b = new Derived;
assert(typeid(*b) == typeid(Derived{})); // OK
```

`typeid` can also be applied to a type directly. In this case, first top-level references are stripped, then top-level cv-qualification is ignored. Thus, the above example could have been written with `typeid(Derived)` instead of `typeid(Derived{})`:

```
assert(typeid(*b) == typeid(Derived{})); // OK
```

If `typeid` is applied to any expression that is *not* of polymorphic class type, the operand is not evaluated, and the type info returned is for the static type.

```
struct Base {
    // note: no virtual destructor
};
struct Derived : Base {};
Derived d;
Base& b = d;
```

```
assert(typeid(b) == typeid(Base)); // not Derived
assert(typeid(std::declval<Base>()) == typeid(Base)); // OK because unevaluated
```

Section 84.3: Name of a type

You can retrieve the implementation defined name of a type in runtime by using the `.name()` member function of the `std::type_info` object returned by `typeid`.

```
#include <iostream>
#include <typeinfo>

int main()
{
    int speed = 110;

    std::cout << typeid(speed).name() << '\n';
}
```

Output (implementation-defined):

```
int
```

Section 84.4: When to use which cast in c++

Use **dynamic_cast** for converting pointers/references within an inheritance hierarchy.

Use **static_cast** for ordinary type conversions.

Use **reinterpret_cast** for low-level reinterpreting of bit patterns. Use with extreme caution.

Use **const_cast** for casting away const/volatile. Avoid this unless you are stuck using a const-incorrect API.

Chapter 85: Mutexes

Section 85.1: Mutex Types

C++1x offers a selection of mutex classes:

- std::mutex - offers simple locking functionality.
- std::timed_mutex - offers try_to_lock functionality
- std::recursive_mutex - allows recursive locking by the same thread.
- std::shared_mutex, std::shared_timed_mutex - offers shared and unique lock functionality.

Section 85.2: std::lock

`std::lock` uses deadlock avoidance algorithms to lock one or more mutexes. If an exception is thrown during a call to lock multiple objects, `std::lock` unlocks the successfully locked objects before re-throwing the exception.

```
std::lock(_mutex1, _mutex2);
```

Section 85.3: std::unique_lock, std::shared_lock, std::lock_guard

Used for the RAII style acquiring of try locks, timed try locks and recursive locks.

`std::unique_lock` allows for exclusive ownership of mutexes.

`std::shared_lock` allows for shared ownership of mutexes. Several threads can hold `std::shared_locks` on a `std::shared_mutex`. Available from C++ 14.

`std::lock_guard` is a lightweight alternative to `std::unique_lock` and `std::shared_lock`.

```
#include <unordered_map>
#include <mutex>
#include <shared_mutex>
#include <thread>
#include <string>
#include <iostream>

class PhoneBook {
public:
    std::string getPhoneNo( const std::string & name )
    {
        std::shared_lock<std::shared_timed_mutex> l(_protect);
        auto it =  _phonebook.find( name );
        if ( it != _phonebook.end() )
            return (*it).second;
        return "";
    }
    void addPhoneNo ( const std::string & name, const std::string & phone )
    {
        std::unique_lock<std::shared_timed_mutex> l(_protect);
        _phonebook[name] = phone;
    }

    std::shared_timed_mutex _protect;
    std::unordered_map<std::string,std::string>  _phonebook;
```

```
};
```

Section 85.4: Strategies for lock classes: std::try_to_lock, std::adopt_lock, std::defer_lock

When creating a std::unique_lock, there are three different locking strategies to choose from: `std::try_to_lock`, `std::defer_lock` and `std::adopt_lock`

1. `std::try_to_lock` allows for trying a lock without blocking:

```
{
    std::atomic_int temp {0};
    std::mutex _mutex;

    std::thread t( [&](){

        while( temp!= -1){
            std::this_thread::sleep_for(std::chrono::seconds(5));
            std::unique_lock<std::mutex> lock( _mutex, std::try_to_lock);

            if(lock.owns_lock()){
                //do something
                temp=0;
            }
        }
    });

    while ( true )
    {
        std::this_thread::sleep_for(std::chrono::seconds(1));
        std::unique_lock<std::mutex> lock( _mutex, std::try_to_lock);
        if(lock.owns_lock()){
            if (temp < INT_MAX){
                ++temp;
            }
            std::cout << temp << std::endl;
        }
    }
}
```

2. `std::defer_lock` allows for creating a lock structure without acquiring the lock. When locking more than one mutex, there is a window of opportunity for a deadlock if two function callers try to acquire the locks at the same time:

```
{
    std::unique_lock<std::mutex> lock1(_mutex1, std::defer_lock);
    std::unique_lock<std::mutex> lock2(_mutex2, std::defer_lock);
    lock1.lock()
    lock2.lock(); // deadlock here
    std::cout << "Locked! << std::endl;
    //...
}
```

With the following code, whatever happens in the function, the locks are acquired and released in appropriate order:

```
   {
       std::unique_lock<std::mutex> lock1(_mutex1, std::defer_lock);
       std::unique_lock<std::mutex> lock2(_mutex2, std::defer_lock);
```

```
        std::lock(lock1,lock2); // no deadlock possible
        std::cout << "Locked! << std::endl;
        //...

    }
```

3. `std::adopt_lock` does not attempt to lock a second time if the calling thread currently owns the lock.

```
{
    std::unique_lock<std::mutex> lock1(_mutex1, std::adopt_lock);
    std::unique_lock<std::mutex> lock2(_mutex2, std::adopt_lock);
    std::cout << "Locked! << std::endl;
    //...
}
```

Something to keep in mind is that std::adopt_lock is not a substitute for recursive mutex usage. When the lock goes out of scope the mutex is **released**.

Section 85.5: std::mutex

std::mutex is a simple, non-recursive synchronization structure that is used to protect data which is accessed by multiple threads.

```
    std::atomic_int temp{0};
    std::mutex _mutex;

    std::thread t( [&](){

                  while( temp!= -1){
                      std::this_thread::sleep_for(std::chrono::seconds(5));
                      std::unique_lock<std::mutex> lock( _mutex);

                          temp=0;
                  }
             });


    while ( true )
    {
        std::this_thread::sleep_for(std::chrono::milliseconds(1));
        std::unique_lock<std::mutex> lock( _mutex, std::try_to_lock);
        if ( temp < INT_MAX )
            temp++;
        cout << temp << endl;

    }
```

Section 85.6: std::scoped_lock (C++ 17)

`std::scoped_lock` provides RAII style semantics for owning one more mutexes, combined with the lock avoidance algorithms used by `std::lock`. When `std::scoped_lock` is destroyed, mutexes are released in the reverse order from which they where acquired.

```
{
    std::scoped_lock lock{_mutex1,_mutex2};
    //do something
}
```

Chapter 86: Recursive Mutex

Section 86.1: std::recursive_mutex

Recursive mutex allows the same thread to recursively lock a resource - up to an unspecified limit.

There are very few real-word justifications for this. Certain complex implementations might need to call an overloaded copy of a function without releasing the lock.

```
std::atomic_int temp{0};
std::recursive_mutex _mutex;

//launch_deferred launches asynchronous tasks on the same thread id

auto future1 = std::async(
            std::launch::deferred,
            [&]()
            {
                std::cout << std::this_thread::get_id() << std::endl;

                std::this_thread::sleep_for(std::chrono::seconds(3));
                std::unique_lock<std::recursive_mutex> lock( _mutex);
                temp=0;

            });

auto future2 = std::async(
            std::launch::deferred,
            [&]()
            {
                std::cout << std::this_thread::get_id() << std::endl;
                while ( true )
                {
                    std::this_thread::sleep_for(std::chrono::milliseconds(1));
                    std::unique_lock<std::recursive_mutex> lock( _mutex, std::try_to_lock);
                    if ( temp < INT_MAX )
                         temp++;

                    cout << temp << endl;

                }
            });
future1.get();
future2.get();
```

Chapter 87: Semaphore

Semaphores are not available in C++ as of now, but can easily be implemented with a mutex and a condition variable.

This example was taken from:

C++0x has no semaphores? How to synchronize threads?

Section 87.1: Semaphore C++ 11

```
#include <mutex>
#include <condition_variable>

class Semaphore {
public:
    Semaphore (int count_ = 0)
    : count(count_)
    {
    }

    inline void notify( int tid ) {
        std::unique_lock<std::mutex> lock(mtx);
        count++;
        cout << "thread " << tid <<  " notify" << endl;
        //notify the waiting thread
        cv.notify_one();
    }
    inline void wait( int tid ) {
        std::unique_lock<std::mutex> lock(mtx);
        while(count == 0) {
            cout << "thread " << tid << " wait" << endl;
            //wait on the mutex until notify is called
            cv.wait(lock);
            cout << "thread " << tid << " run" << endl;
        }
        count--;
    }
private:
    std::mutex mtx;
    std::condition_variable cv;
    int count;
};
```

Section 87.2: Semaphore class in action

The following function adds four threads. Three threads compete for the semaphore, which is set to a count of one. A slower thread calls `notify_one()`, allowing one of the waiting threads to proceed.

The result is that `s1` immediately starts spinning, causing the Semaphore's usage `count` to remain below 1. The other threads wait in turn on the condition variable until notify() is called.

```
int main()
{
    Semaphore sem(1);

    thread s1([&]() {
```

```
        while(true) {
            this_thread::sleep_for(std::chrono::seconds(5));
            sem.wait( 1 );
        }
    });
    thread s2([&]() {
        while(true){
            sem.wait( 2 );
        }
    });
    thread s3([&]() {
        while(true) {
            this_thread::sleep_for(std::chrono::milliseconds(600));
            sem.wait( 3 );
        }
    });
    thread s4([&]() {
        while(true) {
            this_thread::sleep_for(std::chrono::seconds(5));
            sem.notify( 4 );
        }
    });


    s1.join();
    s2.join();
    s3.join();
    s4.join();

    ...
    }
```

Chapter 88: Futures and Promises

Promises and Futures are used to ferry a single object from one thread to another.

A `std::promise` object is set by the thread which generates the result.

A `std::future` object can be used to retrieve a value, to test to see if a value is available, or to halt execution until the value is available.

Section 88.1: Async operation classes

- std::async: performs an asynchronous operation.
- std::future: provides access to the result of an asynchronous operation.
- std::promise: packages the result of an asynchronous operation.
- std::packaged_task: bundles a function and the associated promise for its return type.

Section 88.2: std::future and std::promise

The following example sets a promise to be consumed by another thread:

```
    {
        auto promise = std::promise<std::string>();

        auto producer = std::thread([&]
        {
            promise.set_value("Hello World");
        });

        auto future = promise.get_future();

        auto consumer = std::thread([&]
        {
            std::cout << future.get();
        });

        producer.join();
        consumer.join();
}
```

Section 88.3: Deferred async example

This code implements a version of `std::async`, but it behaves as if `async` were always called with the `deferred` launch policy. This function also does not have `async`'s special `future` behavior; the returned `future` can be destroyed without ever acquiring its value.

```
template<typename F>
auto async_deferred(F&& func) -> std::future<decltype(func())>
{
    using result_type = decltype(func());

    auto promise = std::promise<result_type>();
    auto future  = promise.get_future();

    std::thread(std::bind([=](std::promise<result_type>& promise)
    {
        try
```

```
        {
            promise.set_value(func());
            // Note: Will not work with std::promise<void>. Needs some meta-template programming
which is out of scope for this example.
        }
        catch(...)
        {
            promise.set_exception(std::current_exception());
        }
    }, std::move(promise))).detach();

    return future;
}
```

Section 88.4: std::packaged_task and std::future

`std::packaged_task` bundles a function and the associated promise for its return type:

```
template<typename F>
auto async_deferred(F&& func) -> std::future<decltype(func())>
{
    auto task   = std::packaged_task<decltype(func())()>(std::forward<F>(func));
    auto future = task.get_future();

    std::thread(std::move(task)).detach();

    return std::move(future);
}
```

The thread starts running immediately. We can either detach it, or have join it at the end of the scope. When the function call to std::thread finishes, the result is ready.

Note that this is slightly different from `std::async` where the returned `std::future` when destructed will actually **block** until the thread is finished.

Section 88.5: std::future_error and std::future_errc

If constraints for std::promise and std::future are not met an exception of type std::future_error is thrown.

The error code member in the exception is of type std::future_errc and values are as below, along with some test cases:

```
enum class future_errc {
    broken_promise             = /* the task is no longer shared */,
    future_already_retrieved   = /* the answer was already retrieved */,
    promise_already_satisfied  = /* the answer was stored already */,
    no_state                   = /* access to a promise in non-shared state */
};
```

Inactive promise:

```
int test()
{
    std::promise<int> pr;
    return 0; // returns ok
}
```

Active promise, unused:

```
  int test()
    {
        std::promise<int> pr;
        auto fut = pr.get_future(); //blocks indefinitely!
        return 0;
    }
```

Double retrieval:

```
int test()
{
    std::promise<int> pr;
    auto fut1 = pr.get_future();

    try{
        auto fut2 = pr.get_future();    //   second attempt to get future
        return 0;
    }
    catch(const std::future_error& e)
    {
        cout << e.what() << endl;       //   Error: "The future has already been retrieved from the
promise or packaged_task."
        return -1;
    }
    return fut2.get();
}
```

Setting std::promise value twice:

```
int test()
{
    std::promise<int> pr;
    auto fut = pr.get_future();
    try{
        std::promise<int> pr2(std::move(pr));
        pr2.set_value(10);
        pr2.set_value(10);  // second attempt to set promise throws exception
    }
    catch(const std::future_error& e)
    {
        cout << e.what() << endl;       //   Error: "The state of the promise has already been
set."
        return -1;
    }
    return fut.get();
}
```

Section 88.6: std::future and std::async

In the following naive parallel merge sort example, `std::async` is used to launch multiple parallel merge_sort tasks. `std::future` is used to wait for the results and synchronize them:

```
#include <iostream>
using namespace std;


void merge(int low,int mid,int high, vector<int>&num)
```

```
{
    vector<int> copy(num.size());
    int h,i,j,k;
    h=low;
    i=low;
    j=mid+1;

    while((h<=mid)&&(j<=high))
    {
        if(num[h]<=num[j])
        {
            copy[i]=num[h];
            h++;
        }
        else
        {
            copy[i]=num[j];
            j++;
        }
        i++;
    }
    if(h>mid)
    {
        for(k=j;k<=high;k++)
        {
            copy[i]=num[k];
            i++;
        }
    }
    else
    {
        for(k=h;k<=mid;k++)
        {
            copy[i]=num[k];
            i++;
        }
    }
    for(k=low;k<=high;k++)
        swap(num[k],copy[k]);
}


void merge_sort(int low,int high,vector<int>& num)
{
    int mid;
    if(low<high)
    {
        mid = low + (high-low)/2;
        auto future1    =  std::async(std::launch::deferred,[&]()
                                    {
                                      merge_sort(low,mid,num);
                                    });
        auto future2    =  std::async(std::launch::deferred, [&]()
                                     {
                                        merge_sort(mid+1,high,num) ;
                                     });

        future1.get();
        future2.get();
        merge(low,mid,high,num);
    }
}
```

Note: In the example `std::async` is launched with policy `std::launch_deferred`. This is to avoid a new thread being created in every call. In the case of our example, the calls to `std::async` are made out of order, the they synchronize at the calls for `std::future::get()`.

`std::launch_async` forces a new thread to be created in every call.

The default policy is `std::launch::deferred| std::launch::async`, meaning the implementation determines the policy for creating new threads.

Chapter 89: Atomic Types

Section 89.1: Multi-threaded Access

An atomic type can be used to safely read and write to a memory location shared between two threads.

A Bad example that is likely to cause a data race:

```
#include <thread>
#include <iostream>


//function will add all values including and between 'a' and 'b' to 'result'
void add(int a, int b, int * result) {
    for (int i = a; i <= b; i++) {
        *result += i;
    }
}

int main() {
    //a primitive data type has no thread safety
    int shared = 0;

    //create a thread that may run parallel to the 'main' thread
    //the thread will run the function 'add' defined above with parameters a = 1, b = 100, result =
&shared
    //analogous to 'add(1,100, &shared);'
    std::thread addingThread(add, 1, 100, &shared);

    //attempt to print the value of 'shared' to console
    //main will keep repeating this until the addingThread becomes joinable
    while (!addingThread.joinable()) {
        //this may cause undefined behavior or print a corrupted value
        //if the addingThread tries to write to 'shared' while the main thread is reading it
        std::cout << shared << std::endl;
    }


    //rejoin the thread at the end of execution for cleaning purposes
    addingThread.join();

    return 0;
}
```

The above example may cause a corrupted read and can lead to undefined behavior.

An example with thread safety:

```
#include <atomic>
#include <thread>
#include <iostream>


    //function will add all values including and between 'a' and 'b' to 'result'
void add(int a, int b, std::atomic<int> * result) {
    for (int i = a; i <= b; i++) {
        //atomically add 'i' to result
        result->fetch_add(i);
    }
```

```
}

int main() {
    //atomic template used to store non-atomic objects
    std::atomic<int> shared = 0;

    //create a thread that may run parallel to the 'main' thread
    //the thread will run the function 'add' defined above with parameters a = 1, b = 100, result =
&shared
    //analogous to 'add(1,100, &shared);'
    std::thread addingThread(add, 1, 10000, &shared);

    //print the value of 'shared' to console
    //main will keep repeating this until the addingThread becomes joinable
    while (!addingThread.joinable()) {
        //safe way to read the value of shared atomically for thread safe read
        std::cout << shared.load() << std::endl;
    }


    //rejoin the thread at the end of execution for cleaning purposes
    addingThread.join();

    return 0;
}
```

The above example is safe because all `store()` and `load()` operations of the `atomic` data type protect the encapsulated `int` from simultaneous access.

Chapter 90: Type Erasure

Type erasure is a set of techniques for creating a type that can provide a uniform interface to various underlying types, while hiding the underlying type information from the client. `std::function<R(A...)>`, which has the ability to hold callable objects of various types, is perhaps the best known example of type erasure in C++.

Section 90.1: A move-only `std::function`

`std::function` type erases down to a few operations. One of the things it requires is that the stored value be copyable.

This causes problems in a few contexts, like lambdas storing unique ptrs. If you are using the `std::function` in a context where copying doesn't matter, like a thread pool where you dispatch tasks to threads, this requirement can add overhead.

In particular, `std::packaged_task<Sig>` is a callable object that is move-only. You can store a `std::packaged_task<R(Args...)>` in a `std::packaged_task<void(Args...)>`, but that is a pretty heavy-weight and obscure way to create a move-only callable type-erasure class.

Thus the `task`. This demonstrates how you could write a simple `std::function` type. I omitted the copy constructor (which would involve adding a `clone` method to `details::task_pimpl<...>` as well).

```
template<class Sig>
struct task;

// putting it in a namespace allows us to specialize it nicely for void return value:
namespace details {
  template<class R, class...Args>
  struct task_pimpl {
    virtual R invoke(Args&&...args) const = 0;
    virtual ~task_pimpl() {};
    virtual const std::type_info& target_type() const = 0;
  };

  // store an F.  invoke(Args&&...) calls the f
  template<class F, class R, class...Args>
  struct task_pimpl_impl:task_pimpl<R,Args...> {
    F f;
    template<class Fin>
    task_pimpl_impl( Fin&& fin ):f(std::forward<Fin>(fin)) {}
    virtual R invoke(Args&&...args) const final override {
      return f(std::forward<Args>(args)...);
    }
    virtual const std::type_info& target_type() const final override {
      return typeid(F);
    }
  };

  // the void version discards the return value of f:
  template<class F, class...Args>
  struct task_pimpl_impl<F,void,Args...>:task_pimpl<void,Args...> {
    F f;
    template<class Fin>
    task_pimpl_impl( Fin&& fin ):f(std::forward<Fin>(fin)) {}
    virtual void invoke(Args&&...args) const final override {
      f(std::forward<Args>(args)...);
    }
```

```
    virtual const std::type_info& target_type() const final override {
      return typeid(F);
    }
  };
};

template<class R, class...Args>
struct task<R(Args...)> {
  // semi-regular:
  task()=default;
  task(task&&)=default;
  // no copy

private:
  // aliases to make some SFINAE code below less ugly:
  template<class F>
  using call_r = std::result_of_t<F const&(Args...)>;
  template<class F>
  using is_task = std::is_same<std::decay_t<F>, task>;
public:
  // can be constructed from a callable F
  template<class F,
    // that can be invoked with Args... and converted-to-R:
    class= decltype( (R)(std::declval<call_r<F>>()) ),
    // and is not this same type:
    std::enable_if_t<!is_task<F>{}, int>* = nullptr
  >
  task(F&& f):
    m_pImpl( make_pimpl(std::forward<F>(f)) )
  {}

  // the meat: the call operator
  R operator()(Args... args)const {
        return m_pImpl->invoke( std::forward<Args>(args)... );
  }
  explicit operator bool() const {
    return (bool)m_pImpl;
  }
  void swap( task& o ) {
    std::swap( m_pImpl, o.m_pImpl );
  }
  template<class F>
  void assign( F&& f ) {
    m_pImpl = make_pimpl(std::forward<F>(f));
  }
  // Part of the std::function interface:
  const std::type_info& target_type() const {
    if (!*this) return typeid(void);
    return m_pImpl->target_type();
  }
  template< class T >
  T* target() {
    return target_impl<T>();
  }
  template< class T >
  const T* target() const {
    return target_impl<T>();
  }
  // compare with nullptr    :
  friend bool operator==( std::nullptr_t, task const& self ) { return !self; }
  friend bool operator==( task const& self, std::nullptr_t ) { return !self; }
  friend bool operator!=( std::nullptr_t, task const& self ) { return !!self; }
```

```
  friend bool operator!=( task const& self, std::nullptr_t ) { return !!self; }
private:
  template<class T>
  using pimpl_t = details::task_pimpl_impl<T, R, Args...>;

  template<class F>
  static auto make_pimpl( F&& f ) {
    using dF=std::decay_t<F>;
    using pImpl_t = pimpl_t<dF>;
    return std::make_unique<pImpl_t>(std::forward<F>(f));
  }
  std::unique_ptr<details::task_pimpl<R,Args...>> m_pImpl;

  template< class T >
  T* target_impl() const {
    return dynamic_cast<pimpl_t<T>*>(m_pImpl.get());
  }
};
```

To make this library-worthy, you'd want to add in a small buffer optimization, so it does not store every callable on the heap.

Adding SBO would require a non-default `task(task&&)`, some `std::aligned_storage_t` within the class, a `m_pImpl unique_ptr` with a deleter that can be set to destroy-only (and not return the memory to the heap), and a `emplace_move_to( void* ) = 0` in the `task_pimpl`.

live example of the above code (with no SBO).

Section 90.2: Erasing down to a Regular type with manual vtable

C++ thrives on what is known as a Regular type (or at least Pseudo-Regular).

A Regular type is a type that can be constructed and assigned-to and assigned-from via copy or move, can be destroyed, and can be compared equal-to. It can also be constructed from no arguments. Finally, it also has support for a few other operations that are highly useful in various `std` algorithms and containers.

This is the root paper, but in C++11 would want to add `std::hash` support.

I will use the manual vtable approach to type erasure here.

```
using dtor_unique_ptr = std::unique_ptr<void, void(*)(void*)>;
template<class T, class...Args>
dtor_unique_ptr make_dtor_unique_ptr( Args&&... args ) {
  return {new T(std::forward<Args>(args)...), [](void* self){ delete static_cast<T*>(self); }};
}
struct regular_vtable {
  void(*copy_assign)(void* dest, void const* src); // T&=(T const&)
  void(*move_assign)(void* dest, void* src); // T&=(T&&)
  bool(*equals)(void const* lhs, void const* rhs); // T const&==T const&
  bool(*order)(void const* lhs, void const* rhs); // std::less<T>{}(T const&, T const&)
  std::size_t(*hash)(void const* self); // std::hash<T>{}(T const&)
  std::type_info const&(*type)(); // typeid(T)
  dtor_unique_ptr(*clone)(void const* self); // T(T const&)
};

template<class T>
regular_vtable make_regular_vtable() noexcept {
  return {
```

```
    [](void* dest, void const* src){ *static_cast<T*>(dest) = *static_cast<T const*>(src); },
    [](void* dest, void* src){ *static_cast<T*>(dest) = std::move(*static_cast<T*>(src)); },
    [](void const* lhs, void const* rhs){ return *static_cast<T const*>(lhs) == *static_cast<T
const*>(rhs); },
    [](void const* lhs, void const* rhs) { return std::less<T>{}(*static_cast<T
const*>(lhs),*static_cast<T const*>(rhs)); },
    [](void const* self){ return std::hash<T>{}(*static_cast<T const*>(self)); },
    []()->decltype(auto){ return typeid(T); },
    [](void const* self){ return make_dtor_unique_ptr<T>(*static_cast<T const*>(self)); }
  };
}
template<class T>
regular_vtable const* get_regular_vtable() noexcept {
  static const regular_vtable vtable=make_regular_vtable<T>();
  return &vtable;
}

struct regular_type {
  using self=regular_type;
  regular_vtable const* vtable = 0;
  dtor_unique_ptr ptr{nullptr, [](void*){}};

  bool empty() const { return !vtable; }

  template<class T, class...Args>
  void emplace( Args&&... args ) {
    ptr = make_dtor_unique_ptr<T>(std::forward<Args>(args)...);
    if (ptr)
      vtable = get_regular_vtable<T>();
    else
      vtable = nullptr;
  }
  friend bool operator==(regular_type const& lhs, regular_type const& rhs) {
    if (lhs.vtable != rhs.vtable) return false;
    return lhs.vtable->equals( lhs.ptr.get(), rhs.ptr.get() );
  }
  bool before(regular_type const& rhs) const {
    auto const& lhs = *this;
    if (!lhs.vtable || !rhs.vtable)
      return std::less<regular_vtable const*>{}(lhs.vtable,rhs.vtable);
    if (lhs.vtable != rhs.vtable)
      return lhs.vtable->type().before(rhs.vtable->type());
    return lhs.vtable->order( lhs.ptr.get(), rhs.ptr.get() );
  }
  // technically friend bool operator< that calls before is also required

  std::type_info const* type() const {
    if (!vtable) return nullptr;
    return &vtable->type();
  }
  regular_type(regular_type&& o):
    vtable(o.vtable),
    ptr(std::move(o.ptr))
  {
    o.vtable = nullptr;
  }
  friend void swap(regular_type& lhs, regular_type& rhs){
    std::swap(lhs.ptr, rhs.ptr);
    std::swap(lhs.vtable, rhs.vtable);
  }
  regular_type& operator=(regular_type&& o) {
    if (o.vtable == vtable) {
```

```
      vtable->move_assign(ptr.get(), o.ptr.get());
      return *this;
    }
    auto tmp = std::move(o);
    swap(*this, tmp);
    return *this;
  }
  regular_type(regular_type const& o):
    vtable(o.vtable),
    ptr(o.vtable?o.vtable->clone(o.ptr.get()):dtor_unique_ptr{nullptr, [](void*){}})
  {
    if (!ptr && vtable) vtable = nullptr;
  }
  regular_type& operator=(regular_type const& o) {
    if (o.vtable == vtable) {
      vtable->copy_assign(ptr.get(), o.ptr.get());
      return *this;
    }
    auto tmp = o;
    swap(*this, tmp);
    return *this;
  }
  std::size_t hash() const {
    if (!vtable) return 0;
    return vtable->hash(ptr.get());
  }
  template<class T,
    std::enable_if_t< !std::is_same<std::decay_t<T>, regular_type>{}, int>* =nullptr
  >
  regular_type(T&& t) {
    emplace<std::decay_t<T>>(std::forward<T>(t));
  }
};
namespace std {
  template<>
  struct hash<regular_type> {
    std::size_t operator()( regular_type const& r )const {
      return r.hash();
    }
  };
  template<>
  struct less<regular_type> {
    bool operator()( regular_type const& lhs, regular_type const& rhs ) const {
      return lhs.before(rhs);
    }
  };
}
```

live example.

Such a regular type can be used as a key for a `std::map` or a `std::unordered_map` that accepts *anything regular* for a key, like:

```
std::map<regular_type, std::any>
```

would be basically a map from anothing regular, to anything copyable.

Unlike `any`, my `regular_type` does no small object optimization nor does it support getting the original data back. Getting the original type back isn't hard.

Small object optimization requires that we store an aligned storage buffer within the `regular_type`, and carefully tweak the deleter of the `ptr` to only destroy the object and not delete it.

I would start at `make_dtor_unique_ptr` and teach it how to sometimes store the data in a buffer, and then in the heap if no room in the buffer. That may be sufficient.

Section 90.3: Basic mechanism

Type erasure is a way to hide the type of an object from code using it, even though it is not derived from a common base class. In doing so, it provides a bridge between the worlds of static polymorphism (templates; at the place of use, the exact type must be known at compile time, but it need not be declared to conform to an interface at definition) and dynamic polymorphism (inheritance and virtual functions; at the place of use, the exact type need not be known at compile time, but must be declared to conform to an interface at definition).

The following code shows the basic mechanism of type erasure.

```
#include <ostream>

class Printable
{
public:
  template <typename T>
  Printable(T value) : pValue(new Value<T>(value)) {}
  ~Printable() { delete pValue; }
  void print(std::ostream &os) const { pValue->print(os); }

private:
  Printable(Printable const &)        /* in C++1x: =delete */; // not implemented
  void operator = (Printable const &) /* in C++1x: =delete */; // not implemented
  struct ValueBase
  {
      virtual ~ValueBase() = default;
      virtual void print(std::ostream &) const = 0;
  };
  template <typename T>
  struct Value : ValueBase
  {
      Value(T const &t) : v(t) {}
      virtual void print(std::ostream &os) const { os << v; }
      T v;
  };
  ValueBase *pValue;
};
```

At the use site, only the above definition need to be visible, just as with base classes with virtual functions. For example:

```
#include <iostream>

void print_value(Printable const &p)
{
    p.print(std::cout);
}
```

Note that this is *not* a template, but a normal function that only needs to be declared in a header file, and can be defined in an implementation file (unlike templates, whose definition must be visible at the place of use).

At the definition of the concrete type, nothing needs to be known about `Printable`, it just needs to conform to an

interface, as with templates:

```
struct MyType { int i; };
ostream& operator << (ostream &os, MyType const &mc)
{
  return os << "MyType {" << mc.i << "}";
}
```

We can now pass an object of this class to the function defined above:

```
MyType foo = { 42 };
print_value(foo);
```

Section 90.4: Erasing down to a contiguous buffer of T

Not all type erasure involves virtual inheritance, allocations, placement new, or even function pointers.

What makes type erasure type erasure is that it describes a (set of) behavior(s), and takes any type that supports that behavior and wraps it up. All information that isn't in that set of behaviors is "forgotten" or "erased".

An `array_view` takes its incoming range or container type and erases everything except the fact it is a contiguous buffer of `T`.

```
// helper traits for SFINAE:
template<class T>
using data_t = decltype( std::declval<T>().data() );

template<class Src, class T>
using compatible_data = std::integral_constant<bool, std::is_same< data_t<Src>, T* >{} ||
std::is_same< data_t<Src>, std::remove_const_t<T>* >{}>;

template<class T>
struct array_view {
  // the core of the class:
  T* b=nullptr;
  T* e=nullptr;
  T* begin() const { return b; }
  T* end() const { return e; }

  // provide the expected methods of a good contiguous range:
  T* data() const { return begin(); }
  bool empty() const { return begin()==end(); }
  std::size_t size() const { return end()-begin(); }

  T& operator[](std::size_t i)const{ return begin()[i]; }
  T& front()const{ return *begin(); }
  T& back()const{ return *(end()-1); }

  // useful helpers that let you generate other ranges from this one
  // quickly and safely:
  array_view without_front( std::size_t i=1 ) const {
    i = (std::min)(i, size());
    return {begin()+i, end()};
  }
  array_view without_back( std::size_t i=1 ) const {
    i = (std::min)(i, size());
    return {begin(), end()-i};
  }
```

```
  // array_view is plain old data, so default copy:
  array_view(array_view const&)=default;
  // generates a null, empty range:
  array_view()=default;

  // final constructor:
  array_view(T* s, T* f):b(s),e(f) {}
  // start and length is useful in my experience:
  array_view(T* s, std::size_t length):array_view(s, s+length) {}

  // SFINAE constructor that takes any .data() supporting container
  // or other range in one fell swoop:
  template<class Src,
    std::enable_if_t< compatible_data<std::remove_reference_t<Src>&, T >{}, int>* =nullptr,
    std::enable_if_t< !std::is_same<std::decay_t<Src>, array_view >{}, int>* =nullptr
  >
  array_view( Src&& src ):
    array_view( src.data(), src.size() )
  {}

  // array constructor:
  template<std::size_t N>
  array_view( T(&arr)[N] ):array_view(arr, N) {}

  // initializer list, allowing {} based:
  template<class U,
    std::enable_if_t< std::is_same<const U, T>{}, int>* =nullptr
  >
  array_view( std::initializer_list<U> il ):array_view(il.begin(), il.end()) {}
};
```

an `array_view` takes any container that supports `.data()` returning a pointer to `T` and a `.size()` method, or an array, and erases it down to being a random-access range over contiguous `T`s.

It can take a `std::vector<T>`, a `std::string<T>` a `std::array<T, N>` a `T[37]`, an initializer list (including `{}` based ones), or something else you make up that supports it (via `T* x.data()` and `size_t x.size()`).

In this case, the data we can extract from the thing we are erasing, together with our "view" non-owning state, means we don't have to allocate memory or write custom type-dependent functions.

Live example.

An improvement would be to use a non-member `data` and a non-member `size` in an ADL-enabled context.

Section 90.5: Type erasing type erasure with std::any

This example uses C++14 and `boost::any`. In C++17 you can swap in `std::any` instead.

The syntax we end up with is:

```
const auto print =
  make_any_method<void(std::ostream&)>([](auto&& p, std::ostream& t){ t << p << "\n"; });

super_any<decltype(print)> a = 7;

(a->*print)(std::cout);
```

which is almost optimal.

This example is based off of work by @dyp and @cpplearner as well as my own.

First we use a tag to pass around types:

```
template<class T>struct tag_t{constexpr tag_t(){};};
template<class T>constexpr tag_t<T> tag{};
```

This trait class gets the signature stored with an `any_method`:

This creates a function pointer type, and a factory for said function pointers, given an `any_method`:

```
template<class any_method>
using any_sig_from_method = typename any_method::signature;

template<class any_method, class Sig=any_sig_from_method<any_method>>
struct any_method_function;

template<class any_method, class R, class...Args>
struct any_method_function<any_method, R(Args...)>
{
  template<class T>
  using decorate = std::conditional_t< any_method::is_const, T const, T >;

  using any = decorate<boost::any>;

  using type = R(*)(any&, any_method const*, Args&&...);
  template<class T>
  type operator()( tag_t<T> )const{
    return +[](any& self, any_method const* method, Args&&...args) {
      return (*method)( boost::any_cast<decorate<T>&>(self), decltype(args)(args)... );
    };
  }
};
```

`any_method_function::type` is the type of a function pointer we will store alongside the instance. `any_method_function::operator()` takes a `tag_t<T>` and writes a custom instance of the `any_method_function::type` that assumes the `any&` is going to be a `T`.

We want to be able to type-erase more than one method at a time. So we bundle them up in a tuple, and write a helper wrapper to stick the tuple into static storage on a per-type basis and maintain a pointer to them.

```
template<class...any_methods>
using any_method_tuple = std::tuple< typename any_method_function<any_methods>::type... >;

template<class...any_methods, class T>
any_method_tuple<any_methods...> make_vtable( tag_t<T> ) {
  return std::make_tuple(
    any_method_function<any_methods>{}(tag<T>)...
  );
}

template<class...methods>
struct any_methods {
private:
  any_method_tuple<methods...> const* vtable = 0;
  template<class T>
  static any_method_tuple<methods...> const* get_vtable( tag_t<T> ) {
    static const auto table = make_vtable<methods...>(tag<T>);
    return &table;
```

```
  }
public:
  any_methods() = default;
  template<class T>
  any_methods( tag_t<T> ): vtable(get_vtable(tag<T>)) {}
  any_methods& operator=(any_methods const&)=default;
  template<class T>
  void change_type( tag_t<T> ={} ) { vtable = get_vtable(tag<T>); }

  template<class any_method>
  auto get_invoker( tag_t<any_method> ={} ) const {
    return std::get<typename any_method_function<any_method>::type>( *vtable );
  }
};
```

We could specialize this for a cases where the vtable is small (for example, 1 item), and use direct pointers stored in-class in those cases for efficiency.

Now we start the `super_any`. I use `super_any_t` to make the declaration of `super_any` a bit easier.

```
template<class...methods>
struct super_any_t;
```

This searches the methods that the super any supports for SFINAE and better error messages:

```
template<class super_any, class method>
struct super_method_applies_helper : std::false_type {};

template<class M0, class...Methods, class method>
struct super_method_applies_helper<super_any_t<M0, Methods...>, method> :
    std::integral_constant<bool, std::is_same<M0, method>{}  ||
super_method_applies_helper<super_any_t<Methods...>, method>{}>
{};

template<class...methods, class method>
auto super_method_test( super_any_t<methods...> const&, tag_t<method> )
{
  return std::integral_constant<bool, super_method_applies_helper< super_any_t<methods...>, method
>{} && method::is_const >{};
}
template<class...methods, class method>
auto super_method_test( super_any_t<methods...>&, tag_t<method> )
{
  return std::integral_constant<bool, super_method_applies_helper< super_any_t<methods...>, method
>{} >{};
}

template<class super_any, class method>
struct super_method_applies:
    decltype( super_method_test( std::declval<super_any>(), tag<method> ) )
{};
```

Next we create the `any_method` type. An `any_method` is a pseudo-method-pointer. We create it globally and `const`ly using syntax like:

```
const auto print=make_any_method( [](auto&&self, auto&&os){ os << self; } );
```

or in C++17:

```
const any_method print=[](auto&&self, auto&&os){ os << self; };
```

Note that using a non-lambda can make things hairy, as we use the type for a lookup step. This can be fixed, but would make this example longer than it already is. So always initialize an any method from a lambda, or from a type parametarized on a lambda.

```
template<class Sig, bool const_method, class F>
struct any_method {
  using signature=Sig;
  enum{is_const=const_method};
private:
  F f;
public:

  template<class Any,
    // SFINAE testing that one of the Anys's matches this type:
    std::enable_if_t< super_method_applies< Any&&, any_method >{}, int>* =nullptr
  >
  friend auto operator->*( Any&& self, any_method const& m ) {
    // we don't use the value of the any_method, because each any_method has
    // a unique type (!) and we check that one of the auto*'s in the super_any
    // already has a pointer to us.  We then dispatch to the corresponding
    // any_method_data...

    return [&self, invoke = self.get_invoker(tag<any_method>), m](auto&&...args)->decltype(auto)
    {
      return invoke( decltype(self)(self), &m, decltype(args)(args)... );
    };
  }
  any_method( F fin ):f(std::move(fin)) {}

  template<class...Args>
  decltype(auto) operator()(Args&&...args)const {
    return f(std::forward<Args>(args)...);
  }
};
```

A factory method, not needed in C++17 I believe:

```
template<class Sig, bool is_const=false, class F>
any_method<Sig, is_const, std::decay_t<F>>
make_any_method( F&& f ) {
  return {std::forward<F>(f)};
}
```

This is the augmented `any`. It is both an `any`, and it carries around a bundle of type-erasure function pointers that change whenever the contained `any` does:

```
template<class... methods>
struct super_any_t:boost::any, any_methods<methods...> {
  using vtable=any_methods<methods...>;
public:
  template<class T,
    std::enable_if_t< !std::is_base_of<super_any_t, std::decay_t<T>>{}, int> =0
  >
  super_any_t( T&& t ):
    boost::any( std::forward<T>(t) )
  {
    using dT=std::decay_t<T>;
```

```
    this->change_type( tag<dT> );
  }

  boost::any& as_any()&{return *this;}
  boost::any&& as_any()&&{return std::move(*this);}
  boost::any const& as_any()const&{return *this;}
  super_any_t()=default;
  super_any_t(super_any_t&& o):
    boost::any( std::move( o.as_any() ) ),
    vtable(o)
  {}
  super_any_t(super_any_t const& o):
    boost::any( o.as_any() ),
    vtable(o)
  {}
  template<class S,
    std::enable_if_t< std::is_same<std::decay_t<S>, super_any_t>{}, int> =0
  >
  super_any_t( S&& o ):
    boost::any( std::forward<S>(o).as_any() ),
    vtable(o)
  {}
  super_any_t& operator=(super_any_t&&)=default;
  super_any_t& operator=(super_any_t const&)=default;

  template<class T,
    std::enable_if_t< !std::is_same<std::decay_t<T>, super_any_t>{}, int>* =nullptr
  >
  super_any_t& operator=( T&& t ) {
    ((boost::any&)*this) = std::forward<T>(t);
    using dT=std::decay_t<T>;
    this->change_type( tag<dT> );
    return *this;
  }
};
```

Because we store the any_methods as const objects, this makes making a super_any a bit easier:

```
template<class...Ts>
using super_any = super_any_t< std::remove_cv_t<Ts>... >;
```

Test code:

```
const auto print = make_any_method<void(std::ostream&)>([](auto&& p, std::ostream& t){ t << p <<
"\n"; });
const auto wprint = make_any_method<void(std::wostream&)>([](auto&& p, std::wostream& os ){ os << p
<< L"\n"; });

int main()
{
  super_any<decltype(print), decltype(wprint)> a = 7;
  super_any<decltype(print), decltype(wprint)> a2 = 7;

  (a->*print)(std::cout);
  (a->*wprint)(std::wcout);
}
```

live example.

Originally posted here in a SO self question & answer (and people noted above helped with the implementation).

Chapter 91: Explicit type conversions

An expression can be *explicitly converted* or *cast* to type `T` using `dynamic_cast<T>`, `static_cast<T>`, `reinterpret_cast<T>`, or `const_cast<T>`, depending on what type of cast is intended.

C++ also supports function-style cast notation, `T(expr)`, and C-style cast notation, `(T)expr`.

Section 91.1: C-style casting

C-Style casting can be considered 'Best effort' casting and is named so as it is the only cast which could be used in C. The syntax for this cast is `(NewType)variable`.

Whenever this cast is used, it uses one of the following c++ casts (in order):

- `const_cast<NewType>(variable)`
- `static_cast<NewType>(variable)`
- `const_cast<NewType>(static_cast<const NewType>(variable))`
- `reinterpret_cast<const NewType>(variable)`
- `const_cast<NewType>(reinterpret_cast<const NewType>(variable))`

Functional casting is very similar, though as a few restrictions as the result of its syntax: `NewType(expression)`. As a result, only types without spaces can be cast to.

It's better to use new c++ cast, because s more readable and can be spotted easily anywhere inside a C++ source code and errors will be detected in compile-time, instead in run-time.

As this cast can result in unintended `reinterpret_cast`, it is often considered dangerous.

Section 91.2: Casting away constness

A pointer to a const object can be converted to a pointer to non-const object using the `const_cast` keyword. Here we use `const_cast` to call a function that is not const-correct. It only accepts a non-const `char*` argument even though it never writes through the pointer:

```
void bad_strlen(char*);
const char* s = "hello, world!";
bad_strlen(s);                    // compile error
bad_strlen(const_cast<char*>(s)); // OK, but it's better to make bad_strlen accept const char*
```

`const_cast` to reference type can be used to convert a const-qualified lvalue into a non-const-qualified value.

`const_cast` is dangerous because it makes it impossible for the C++ type system to prevent you from trying to modify a const object. Doing so results in undefined behavior.

```
const int x = 123;
int& mutable_x = const_cast<int&>(x);
mutable_x = 456; // may compile, but produces *undefined behavior*
```

Section 91.3: Base to derived conversion

A pointer to base class can be converted to a pointer to derived class using `static_cast`. `static_cast` does not do any run-time checking and can lead to undefined behaviour when the pointer does not actually point to the desired type.

```
struct Base {};
struct Derived : Base {};
Derived d;
Base* p1 = &d;
Derived* p2 = p1;                        // error; cast required
Derived* p3 = static_cast<Derived*>(p1); // OK; p2 now points to Derived object
Base b;
Base* p4 = &b;
Derived* p5 = static_cast<Derived*>(p4); // undefined behaviour since p4 does not
                                         // point to a Derived object
```

Likewise, a reference to base class can be converted to a reference to derived class using `static_cast`.

```
struct Base {};
struct Derived : Base {};
Derived d;
Base& r1 = d;
Derived& r2 = r1;                        // error; cast required
Derived& r3 = static_cast<Derived&>(r1); // OK; r3 now refers to Derived object
```

If the source type is polymorphic, `dynamic_cast` can be used to perform a base to derived conversion. It performs a run-time check and failure is recoverable instead of producing undefined behaviour. In the pointer case, a null pointer is returned upon failure. In the reference case, an exception is thrown upon failure of type `std::bad_cast` (or a class derived from `std::bad_cast`).

```
struct Base { virtual ~Base(); }; // Base is polymorphic
struct Derived : Base {};
Base* b1 = new Derived;
Derived* d1 = dynamic_cast<Derived*>(b1); // OK; d1 points to Derived object
Base* b2 = new Base;
Derived* d2 = dynamic_cast<Derived*>(b2); // d2 is a null pointer
```

Section 91.4: Conversion between pointer and integer

An object pointer (including `void*`) or function pointer can be converted to an integer type using `reinterpret_cast`. This will only compile if the destination type is long enough. The result is implementation-defined and typically yields the numeric address of the byte in memory that the pointer pointers to.

Typically, `long` or `unsigned long` is long enough to hold any pointer value, but this is not guaranteed by the standard.

Version ≥ C++11

If the types `std::intptr_t` and `std::uintptr_t` exist, they are guaranteed to be long enough to hold a `void*` (and hence any pointer to object type). However, they are not guaranteed to be long enough to hold a function pointer.

Similarly, `reinterpret_cast` can be used to convert an integer type into a pointer type. Again the result is implementation-defined, but a pointer value is guaranteed to be unchanged by a round trip through an integer type. The standard does not guarantee that the value zero is converted to a null pointer.

```
void register_callback(void (*fp)(void*), void* arg); // probably a C API
void my_callback(void* x) {
    std::cout << "the value is: " << reinterpret_cast<long>(x); // will probably compile
}
long x;
std::cin >> x;
```

```
register_callback(my_callback,
                  reinterpret_cast<void*>(x)); // hopefully this doesn't lose information...
```

Section 91.5: Conversion by explicit constructor or explicit conversion function

A conversion that involves calling an explicit constructor or conversion function can't be done implicitly. We can request that the conversion be done explicitly using `static_cast`. The meaning is the same as that of a direct initialization, except that the result is a temporary.

```
class C {
    std::unique_ptr<int> p;
  public:
    explicit C(int* p) : p(p) {}
};
void f(C c);
void g(int* p) {
    f(p);                 // error: C::C(int*) is explicit
    f(static_cast<C>(p)); // ok
    f(C(p));              // equivalent to previous line
    C c(p); f(c);         // error: C is not copyable
}
```

Section 91.6: Implicit conversion

`static_cast` can perform any implicit conversion. This use of `static_cast` can occasionally be useful, such as in the following examples:

- When passing arguments to an ellipsis, the "expected" argument type is not statically known, so no implicit conversion will occur.

  ```
  const double x = 3.14;
  printf("%d\n", static_cast<int>(x)); // prints 3
  // printf("%d\n", x); // undefined behaviour; printf is expecting an int here
  // alternative:
  // const int y = x; printf("%d\n", y);
  ```

  Without the explicit type conversion, a `double` object would be passed to the ellipsis, and undefined behaviour would occur.

- A derived class assignment operator can call a base class assignment operator like so:

  ```
  struct Base { /* ... */ };
  struct Derived : Base {
      Derived& operator=(const Derived& other) {
          static_cast<Base&>(*this) = other;
          // alternative:
          // Base& this_base_ref = *this; this_base_ref = other;
      }
  };
  ```

Section 91.7: Enum conversions

`static_cast` can convert from an integer or floating point type to an enumeration type (whether scoped or

unscoped), and *vice versa.* It can also convert between enumeration types.

- The conversion from an unscoped enumeration type to an arithmetic type is an implicit conversion; it is possible, but not necessary, to use `static_cast`.

Version ≥ C++11

- When a scoped enumeration type is converted to an arithmetic type:

    - If the enum's value can be represented exactly in the destination type, the result is that value.
    - Otherwise, if the destination type is an integer type, the result is unspecified.
    - Otherwise, if the destination type is a floating point type, the result is the same as that of converting to the underlying type and then to the floating point type.

  Example:

```
enum class Format {
    TEXT = 0,
    PDF = 1000,
    OTHER = 2000,
};
Format f = Format::PDF;
int a = f;                         // error
int b = static_cast<int>(f);       // ok; b is 1000
char c = static_cast<char>(f);     // unspecified, if 1000 doesn't fit into char
double d = static_cast<double>(f); // d is 1000.0... probably
```

- When an integer or enumeration type is converted to an enumeration type:

    - If the original value is within the destination enum's range, the result is that value. Note that this value might be unequal to all enumerators.
    - Otherwise, the result is unspecified (<= C++14) or undefined (>= C++17).

  Example:

```
enum Scale {
    SINGLE = 1,
    DOUBLE = 2,
    QUAD = 4
};
Scale s1 = 1;                     // error
Scale s2 = static_cast<Scale>(2); // s2 is DOUBLE
Scale s3 = static_cast<Scale>(3); // s3 has value 3, and is not equal to any enumerator
Scale s9 = static_cast<Scale>(9); // unspecified value in C++14; UB in C++17
```

Version ≥ C++11

- When a floating point type is converted to an enumeration type, the result is the same as converting to the enum's underlying type and then to the enum type.

```
enum Direction {
    UP = 0,
    LEFT = 1,
    DOWN = 2,
    RIGHT = 3,
};
```

```
Direction d = static_cast<Direction>(3.14); // d is RIGHT
```

Section 91.8: Derived to base conversion for pointers to members

A pointer to member of derived class can be converted to a pointer to member of base class using `static_cast`. The types pointed to must match.

If the operand is a null pointer to member value, the result is also a null pointer to member value.

Otherwise, the conversion is only valid if the member pointed to by the operand actually exists in the destination class, or if the destination class is a base or derived class of the class containing the member pointed to by the operand. `static_cast` does not check for validity. If the conversion is not valid, the behaviour is undefined.

```
struct A {};
struct B { int x; };
struct C : A, B { int y; double z; };
int B::*p1 = &B::x;
int C::*p2 = p1;                              // ok; implicit conversion
int B::*p3 = p2;                              // error
int B::*p4 = static_cast<int B::*>(p2);       // ok; p4 is equal to p1
int A::*p5 = static_cast<int A::*>(p2);       // undefined; p2 points to x, which is a member
                                              // of the unrelated class B
double C::*p6 = &C::z;
double A::*p7 = static_cast<double A::*>(p6); // ok, even though A doesn't contain z
int A::*p8 = static_cast<int A::*>(p6);       // error: types don't match
```

Section 91.9: void* to T*

In C++, `void*` cannot be implicitly converted to `T*` where `T` is an object type. Instead, `static_cast` should be used to perform the conversion explicitly. If the operand actually points to a `T` object, the result points to that object. Otherwise, the result is unspecified.

Version ≥ C++11

Even if the operand does not point to a `T` object, as long as the operand points to a byte whose address is properly aligned for the type `T`, the result of the conversion points to the same byte.

```
// allocating an array of 100 ints, the hard way
int* a = malloc(100*sizeof(*a));                    // error; malloc returns void*
int* a = static_cast<int*>(malloc(100*sizeof(*a))); // ok
// int* a = new int[100];                           // no cast needed
// std::vector<int> a(100);                         // better

const char c = '!';
const void* p1 = &c;
const char* p2 = p1;                           // error
const char* p3 = static_cast<const char*>(p1); // ok; p3 points to c
const int* p4 = static_cast<const int*>(p1);   // unspecified in C++03;
                                               // possibly unspecified in C++11 if
                                               // alignof(int) > alignof(char)
char* p5 = static_cast<char*>(p1);             // error: casting away constness
```

Section 91.10: Type punning conversion

A pointer (resp. reference) to an object type can be converted to a pointer (resp. reference) to any other object type using `reinterpret_cast`. This does not call any constructors or conversion functions.

```
int x = 42;
char* p = static_cast<char*>(&x);      // error: static_cast cannot perform this conversion
char* p = reinterpret_cast<char*>(&x); // OK
*p = 'z';                              // maybe this modifies x (see below)
```

Version ≥ C++11

The result of `reinterpret_cast` represents the same address as the operand, provided that the address is appropriately aligned for the destination type. Otherwise, the result is unspecified.

```
int x = 42;
char& r = reinterpret_cast<char&>(x);
const void* px = &x;
const void* pr = &r;
assert(px == pr); // should never fire
```

Version < C++11

The result of `reinterpret_cast` is unspecified, except that a pointer (resp. reference) will survive a round trip from the source type to the destination type and back, as long as the destination type's alignment requirement is not stricter than that of the source type.

```
int x = 123;
unsigned int& r1 = reinterpret_cast<unsigned int&>(x);
int& r2 = reinterpret_cast<int&>(r1);
r2 = 456; // sets x to 456
```

On most implementations, `reinterpret_cast` does not change the address, but this requirement was not standardized until C++11.

`reinterpret_cast` can also be used to convert from one pointer-to-data-member type to another, or one pointer-to-member-function type to another.

Use of `reinterpret_cast` is considered dangerous because reading or writing through a pointer or reference obtained using `reinterpret_cast` may trigger undefined behaviour when the source and destination types are unrelated.

Chapter 92: Unnamed types

Section 92.1: Unnamed classes

Unlike a named class or struct, unnamed classes and structs must be instantiated where they are defined, and cannot have constructors or destructors.

```
struct {
    int foo;
    double bar;
} foobar;

foobar.foo = 5;
foobar.bar = 4.0;

class {
    int baz;
public:
    int buzz;

    void setBaz(int v) {
        baz = v;
    }
} barbar;

barbar.setBaz(15);
barbar.buzz = 2;
```

Section 92.2: As a type alias

Unnamed class types may also be used when creating type aliases, i.e. via `typedef` and `using`:

Version < C++11

```
using vec2d = struct {
    float x;
    float y;
};
```

```
typedef struct {
    float x;
    float y;
} vec2d;
```

```
vec2d pt;
pt.x = 4.f;
pt.y = 3.f;
```

Section 92.3: Anonymous members

As a non-standard extension to C++, common compilers allow the use of classes as anonymous members.

```
struct Example {
    struct {
        int inner_b;
    };

    int outer_b;
```

```
    //The anonymous struct's members are accessed as if members of the parent struct
    Example() : inner_b(2), outer_b(4) {
        inner_b = outer_b + 2;
    }
};

Example ex;

//The same holds true for external code referencing the struct
ex.inner_b -= ex.outer_b;
```

Section 92.4: Anonymous Union

Member names of an anonymous union belong to the scope of the union declaration an must be distinct to all other names of this scope. The example here has the same construction as example Anonymous Members using "struct" but is standard conform.

```
struct Sample {
    union {
        int a;
        int b;
    };
    int c;
};
int main()
{
  Sample sa;
  sa.a =3;
  sa.b =4;
  sa.c =5;
}
```

Chapter 93: Type Traits

Section 93.1: Type Properties

Version ≥ C++11

Type properties compare the modifiers that can be placed upon different variables. The usefulness of these type traits is not always obvious.

Note: The example below would only offer an improvement on a non-optimizing compiler. It is a simple a proof of concept, rather than complex example.

e.g. Fast divide by four.

```
template<typename T>
inline T FastDivideByFour(cont T &var) {
  // Will give an error if the inputted type is not an unsigned integral type.
  static_assert(std::is_unsigned<T>::value && std::is_integral<T>::value,
    "This function is only designed for unsigned integral types.");
  return (var >> 2);
}
```

Is Constant:

This will evaluate as true when type is constant.

```
std::cout << std::is_const<const int>::value << "\n"; // Prints true.
std::cout << std::is_const<int>::value << "\n"; // Prints false.
```

Is Volatile:

This will evaluate as true when the type is volatile.

```
std::cout << std::is_volatile<static volatile int>::value << "\n"; // Prints true.
std::cout << std::is_const<const int>::value << "\n"; // Prints false.
```

Is signed:

This will evaluate as true for all signed types.

```
std::cout << std::is_signed<int>::value << "\n"; // Prints true.
std::cout << std::is_signed<float>::value << "\n"; // Prints true.
std::cout << std::is_signed<unsigned int>::value << "\n"; // Prints false.
std::cout << std::is_signed<uint8_t>::value << "\n"; // Prints false.
```

Is Unsigned:

Will evaluate as true for all unsigned types.

```
std::cout << std::is_unsigned<unsigned int>::value << "\n"; // Prints true.
std::cout << std::is_signed<uint8_t>::value << "\n"; // Prints true.
std::cout << std::is_unsigned<int>::value << "\n"; // Prints false.
std::cout << std::is_signed<float>::value << "\n"; // Prints false.
```

Section 93.2: Standard type traits

Version ≥ C++11

The `type_traits` header contains a set of template classes and helpers to transform and check properties of types at compile-time.

These traits are typically used in templates to check for user errors, support generic programming, and allow for optimizations.

Most type traits are used to check if a type fulfils some criteria. These have the following form:

```
template <class T> struct is_foo;
```

If the template class is instantiated with a type which fulfils some criteria `foo`, then `is_foo<T>` inherits from `std::integral_constant<bool,true>` (a.k.a. `std::true_type`), otherwise it inherits from `std::integral_constant<bool,false>` (a.k.a. `std::false_type`). This gives the trait the following members:

Constants

`static constexpr bool value`

`true` if `T` fulfils the criteria `foo`, `false` otherwise

Functions

`operator bool`

Returns `value`

Version ≥ C++14

`bool operator()`

Returns `value`

Types

Name	Definition
`value_type`	`bool`
`type`	`std::integral_constant<bool,value>`

The trait can then be used in constructs such as `static_assert` or `std::enable_if`. An example with `std::is_pointer`:

```
template <typename T>
void i_require_a_pointer (T t) {
    static_assert(std::is_pointer<T>::value, "T must be a pointer type");
}

//Overload for when T is not a pointer type
template <typename T>
typename std::enable_if<!std::is_pointer<T>::value>::type
does_something_special_with_pointer (T t) {
    //Do something boring
}

//Overload for when T is a pointer type
template <typename T>
```

```
typename std::enable_if<std::is_pointer<T>::value>::type
does_something_special_with_pointer (T t) {
    //Do something special
}
```

There are also various traits which transform types, such as `std::add_pointer` and `std::underlying_type`. These traits generally expose a single `type` member type which contains the transformed type. For example, `std::add_pointer<int>::type` is `int*`.

Section 93.3: Type relations with std::is_same<T, T>

Version ≥ C++11

The `std::is_same<T, T>` type relation is used to compare two types. It will evaluate as boolean, true if the types are the same and false if otherwise.

e.g.

```
// Prints true on most x86 and x86_64 compilers.
std::cout << std::is_same<int, int32_t>::value << "\n";
// Prints false on all compilers.
std::cout << std::is_same<float, int>::value << "\n";
// Prints false on all compilers.
std::cout  << std::is_same<unsigned int, int>::value << "\n";
```

The `std::is_same` type relation will also work regardless of typedefs. This is actually demonstrated in the first example when comparing `int == int32_t` however this is not entirely clear.

e.g.

```
// Prints true on all compilers.
typedef int MyType
std::cout << std::is_same<int, MyType>::value <<  "\n";
```

Using `std::is_same` to warn when improperly using a templated class or function.

When combined with a static assert the `std::is_same` template can be valuable tool in enforcing proper usage of templated classes and functions.

e.g. A function that only allows input from an `int` and a choice of two structs.

```
#include <type_traits>
struct foo {
  int member;
  // Other variables
};

struct bar {
  char member;
};

template<typename T>
int AddStructMember(T var1, int var2) {
  // If type T != foo || T != bar then show error message.
  static_assert(std::is_same<T, foo>::value ||
    std::is_same<T, bar>::value,
    "This function does not support the specified type.");
```

```
  return var1.member + var2;
}
```

Section 93.4: Fundamental type traits

Version ≥ C++11

There are a number of different type traits that compare more general types.

Is Integral:

Evaluates as true for all integer types `int`, `char`, `long`, `unsigned int` etc.

```
std::cout << std::is_integral<int>::value << "\n"; // Prints true.
std::cout << std::is_integral<char>::value << "\n"; // Prints true.
std::cout << std::is_integral<float>::value << "\n"; // Prints false.
```

Is Floating Point:

Evaluates as true for all floating point types. `float`,`double`, `long double` etc.

```
std::cout << std::is_floating_point<float>::value << "\n"; // Prints true.
std::cout << std::is_floating_point<double>::value << "\n"; // Prints true.
std::cout << std::is_floating_point<char>::value << "\n"; // Prints false.
```

Is Enum:

Evaluates as true for all enumerated types, including `enum class`.

```
enum fruit {apple, pair, banana};
enum class vegetable {carrot, spinach, leek};
std::cout << std::is_enum<fruit>::value << "\n"; // Prints true.
std::cout << std::is_enum<vegetable>::value << "\n"; // Prints true.
std::cout << std::is_enum<int>::value << "\n"; // Prints false.
```

Is Pointer:

Evaluates as true for all pointers.

```
std::cout << std::is_pointer<int *>::value << "\n"; // Prints true.
typedef int* MyPTR;
std::cout << std::is_pointer<MyPTR>::value << "\n"; // Prints true.
std::cout << std::is_pointer<int>::value << "\n"; // Prints false.
```

Is Class:

Evaluates as true for all classes and struct, with the exception of `enum class`.

```
struct FOO {int x, y;};
class BAR {
 public:
  int x, y;
};
enum class fruit {apple, pair, banana};
std::cout << std::is_class<FOO>::value << "\n"; // Prints true.
std::cout << std::is_class<BAR>::value << "\n"; // Prints true.
std::cout << std::is_class<fruit>::value << "\n"; // Prints false.
```

```
std::cout << std::is_class<int>::value << "\n"; // Prints false.
```

Chapter 94: Return Type Covariance

Section 94.1: Covariant result version of the base example, static type checking

```
// 2. Covariant result version of the base example, static type checking.

class Top
{
public:
    virtual Top* clone() const = 0;
    virtual ~Top() = default;       // Necessary for `delete` via Top*.
};

class D : public Top
{
public:
    D* /* ← Covariant return */ clone() const override
    { return new D( *this ); }
};

class DD : public D
{
private:
    int answer_ = 42;

public:
    int answer() const
    { return answer_;}

    DD* /* ← Covariant return */ clone() const override
    { return new DD( *this ); }
};

#include <iostream>
using namespace std;

int main()
{
    DD* p1 = new DD();
    DD* p2 = p1->clone();
    // Correct dynamic type DD for *p2 is guaranteed by the static type checking.

    cout << p2->answer() << endl;          // "42"
    delete p2;
    delete p1;
}
```

Section 94.2: Covariant smart pointer result (automated cleanup)

```
// 3. Covariant smart pointer result (automated cleanup).

#include <memory>
using std::unique_ptr;

template< class Type >
auto up( Type* p ) { return unique_ptr<Type>( p ); }
```

```
class Top
{
private:
    virtual Top* virtual_clone() const = 0;

public:
    unique_ptr<Top> clone() const
    { return up( virtual_clone() ); }

    virtual ~Top() = default;        // Necessary for `delete` via Top*.
};

class D : public Top
{
private:
    D* /* ← Covariant return */ virtual_clone() const override
    { return new D( *this ); }

public:
    unique_ptr<D> /* ← Apparent covariant return */ clone() const
    { return up( virtual_clone() ); }
};

class DD : public D
{
private:
    int answer_ = 42;

    DD* /* ← Covariant return */ virtual_clone() const override
    { return new DD( *this ); }

public:
    int answer() const
    { return answer_;}

    unique_ptr<DD> /* ← Apparent covariant return */ clone() const
    { return up( virtual_clone() ); }
};

#include <iostream>
using namespace std;

int main()
{
    auto  p1 = unique_ptr<DD>(new DD());
    auto  p2 = p1->clone();
    // Correct dynamic type DD for *p2 is guaranteed by the static type checking.

    cout << p2->answer() << endl;          // "42"
    // Cleanup is automated via unique_ptr.
 }
```

Chapter 95: Layout of object types

Section 95.1: Class types

By "class", we mean a type that was defined using the `class` or `struct` keyword (but not `enum struct` or `enum class`).

- Even an empty class still occupies at least one byte of storage; it will therefore consist purely of padding. This ensures that if `p` points to an object of an empty class, then `p + 1` is a distinct address and points to a distinct object. However, it is possible for an empty class to have a size of 0 when used as a base class. See empty base optimisation.

```
class Empty_1 {};                               // sizeof(Empty_1)       == 1
class Empty_2 {};                               // sizeof(Empty_2)       == 1
class Derived : Empty_1 {};                     // sizeof(Derived)       == 1
class DoubleDerived : Empty_1, Empty_2 {};      // sizeof(DoubleDerived) == 1
class Holder { Empty_1 e; };                    // sizeof(Holder)        == 1
class DoubleHolder { Empty_1 e1; Empty_2 e2; }; // sizeof(DoubleHolder)  == 2
class DerivedHolder : Empty_1 { Empty_1 e; };   // sizeof(DerivedHolder) == 2
```

- The object representation of a class type contains the object representations of the base class and non-static member types. Therefore, for example, in the following class:

```
struct S {
    int x;
    char* y;
};
```

there is a consecutive sequence of `sizeof(int)` bytes within an `S` object, called a *subobject,* that contain the value of `x`, and another subobject with `sizeof(char*)` bytes that contains the value of `y`. The two cannot be interleaved.

- If a class type has members and/or base classes with types `t1, t2,...tN`, the size must be at least `sizeof(t1) + sizeof(t2) + ... + sizeof(tN)` given the preceding points. However, depending on the alignment requirements of the members and base classes, the compiler may be forced to insert padding between subobjects, or at the beginning or end of the complete object.

```
struct AnInt      { int i; };
  // sizeof(AnInt)        == sizeof(int)
  // Assuming a typical 32- or 64-bit system, sizeof(AnInt)        == 4 (4).
struct TwoInts    { int i, j; };
  // sizeof(TwoInts)      >= 2 * sizeof(int)
  // Assuming a typical 32- or 64-bit system, sizeof(TwoInts)      == 8 (4 + 4).
struct IntAndChar { int i; char c; };
  // sizeof(IntAndChar)   >= sizeof(int) + sizeof(char)
  // Assuming a typical 32- or 64-bit system, sizeof(IntAndChar)   == 8 (4 + 1 + padding).
struct AnIntDerived : AnInt { long long l; };
  // sizeof(AnIntDerived) >= sizeof(AnInt) + sizeof(long long)
  // Assuming a typical 32- or 64-bit system, sizeof(AnIntDerived) == 16 (4 + padding + 8).
```

- If padding is inserted in an object due to alignment requirements, the size will be greater than the sum of the sizes of the members and base classes. With `n`-byte alignment, size will typically be the smallest multiple of `n` which is larger than the size of all members & base classes. Each member `memN` will typically be placed at an address which is a multiple of `alignof(memN)`, and `n` will typically be the largest `alignof` out of all members'

alignofs. Due to this, if a member with a smaller alignof is followed by a member with a larger alignof, there is a possibility that the latter member will not be aligned properly if placed immediately after the former. In this case, padding (also known as an *alignment member*) will be placed between the two members, such that the latter member can have its desired alignment. Conversely, if a member with a larger alignof is followed by a member with a smaller alignof, no padding will usually be necessary. This process is also known as "packing".
Due to classes typically sharing the alignof of their member with the largest alignof, classes will typically be aligned to the alignof of the largest built-in type they directly or indirectly contain.

```
// Assume sizeof(short) == 2, sizeof(int) == 4, and sizeof(long long) == 8.
// Assume 4-byte alignment is specified to the compiler.
struct Char { char c; };
  // sizeof(Char)                == 1 (sizeof(char))
struct Int  { int i; };
  // sizeof(Int)                 == 4 (sizeof(int))
struct CharInt { char c; int i; };
  // sizeof(CharInt)             == 8 (1 (char) + 3 (padding) + 4 (int))
struct ShortIntCharInt { short s; int i; char c; int j; };
  // sizeof(ShortIntCharInt)     == 16 (2 (short) + 2 (padding) + 4 (int) + 1 (char) +
  //                                   3 (padding) + 4 (int))
struct ShortIntCharCharInt { short s; int i; char c; char d; int j; };
  // sizeof(ShortIntCharCharInt) == 16 (2 (short) + 2 (padding) + 4 (int) + 1 (char) +
  //                                   1 (char) + 2 (padding) + 4 (int))
struct ShortCharShortInt { short s; char c; short t; int i; };
  // sizeof(ShortCharShortInt)   == 12 (2 (short) + 1 (char) + 1 (padding) + 2 (short) +
  //                                   2 (padding) + 4 (int))
struct IntLLInt { int i; long long l; int j; };
  // sizeof(IntLLInt)            == 16 (4 (int) + 8 (long long) + 4 (int))
  // If packing isn't explicitly specified, most compilers will pack this as
  //   8-byte alignment, such that:
  // sizeof(IntLLInt)            == 24 (4 (int) + 4 (padding) + 8 (long long) +
  //                                   4 (int) + 4 (padding))

// Assume sizeof(bool) == 1, sizeof(ShortIntCharInt) == 16, and sizeof(IntLLInt) == 24.
// Assume default alignment: alignof(ShortIntCharInt) == 4, alignof(IntLLInt) == 8.
struct ShortChar3ArrShortInt {
    short s;
    char c3[3];
    short t;
    int i;
};
  // ShortChar3ArrShortInt has 4-byte alignment: alignof(int) >= alignof(char) &&
  //                                              alignof(int) >= alignof(short)
  // sizeof(ShortChar3ArrShortInt) == 12 (2 (short) + 3 (char[3]) + 1 (padding) +
  //                                     2 (short) + 4 (int))
  // Note that t is placed at alignment of 2, not 4.  alignof(short) == 2.

struct Large_1 {
    ShortIntCharInt sici;
    bool b;
    ShortIntCharInt tjdj;
};
  // Large_1 has 4-byte alignment.
    // alignof(ShortIntCharInt) == alignof(int) == 4
    // alignof(b) == 1
    // Therefore, alignof(Large_1) == 4.
  // sizeof(Large_1) == 36 (16 (ShortIntCharInt) + 1 (bool) + 3 (padding) +
  //                        16 (ShortIntCharInt))
struct Large_2 {
    IntLLInt illi;
```

```
    float f;
    IntLLInt jmmj;
};
  // Large_2 has 8-byte alignment.
    // alignof(IntLLInt) == alignof(long long) == 8
    // alignof(float) == 4
    // Therefore, alignof(Large_2) == 8.
  // sizeof(Large_2) == 56 (24 (IntLLInt) + 4 (float) + 4 (padding) + 24 (IntLLInt))
```

Version ≥ C++11

- If strict alignment is forced with `alignas`, padding will be used to force the type to meet the specified alignment, even when it would otherwise be smaller. For example, with the definition below, `Chars<5>` will have three (or possibly more) padding bytes inserted at the end so that its total size is 8. It is not possible for a class with an alignment of 4 to have a size of 5 because it would be impossible to make an array of that class, so the size must be "rounded up" to a multiple of 4 by inserting padding bytes.

```
// This type shall always be aligned to a multiple of 4.  Padding shall be inserted as
// needed.
// Chars<1>..Chars<4> are 4 bytes, Chars<5>..Chars<8> are 8 bytes, etc.
template<size_t SZ>
struct alignas(4) Chars { char arr[SZ]; };

static_assert(sizeof(Chars<1>) == sizeof(Chars<4>), "Alignment is strict.\n");
```

- If two non-static members of a class have the same access specifier, then the one that comes later in declaration order is guaranteed to come later in the object representation. But if two non-static members have different access specifiers, their relative order within the object is unspecified.
- It is unspecified what order the base class subobjects appear in within an object, whether they occur consecutively, and whether they appear before, after, or between member subobjects.

Section 95.2: Arithmetic types

Narrow character types

The `unsigned char` type uses all bits to represent a binary number. Therefore, for example, if `unsigned char` is 8 bits long, then the 256 possible bit patterns of a `char` object represent the 256 different values {0, 1, ..., 255}. The number 42 is guaranteed to be represented by the bit pattern `00101010`.

The `signed char` type has no padding bits, *i.e.,* if `signed char` is 8 bits long, then it has 8 bits of capacity to represent a number.

Note that these guarantees do not apply to types other than narrow character types.

Integer types

The unsigned integer types use a pure binary system, but may contain padding bits. For example, it is possible (though unlikely) for `unsigned int` to be 64 bits long but only be capable of storing integers between 0 and 232 - 1, inclusive. The other 32 bits would be padding bits, which should not be written to directly.

The signed integer types use a binary system with a sign bit and possibly padding bits. Values that belong to the common range of a signed integer type and the corresponding unsigned integer type have the same representation. For example, if the bit pattern `0001010010101011` of an `unsigned short` object represents the value `5291`, then it also represents the value `5291` when interpreted as a `short` object.

It is implementation-defined whether a two's complement, one's complement, or sign-magnitude representation is used, since all three systems satisfy the requirement in the previous paragraph.

Floating point types

The value representation of floating point types is implementation-defined. Most commonly, the `float` and `double` types conform to IEEE 754 and are 32 and 64 bits long (so, for example, `float` would have 23 bits of precision which would follow 8 exponent bits and 1 sign bit). However, the standard does not guarantee anything. Floating point types often have "trap representations", which cause errors when they are used in calculations.

Section 95.3: Arrays

An array type has no padding in between its elements. Therefore, an array with element type `T` is just a sequence of `T` objects laid out in memory, in order.

A multidimensional array is an array of arrays, and the above applies recursively. For example, if we have the declaration

```
int a[5][3];
```

then `a` is an array of 5 arrays of 3 `int`s. Therefore, `a[0]`, which consists of the three elements `a[0][0]`, `a[0][1]`, `a[0][2]`, is laid out in memory before `a[1]`, which consists of `a[1][0]`, `a[1][1]`, and `a[1][2]`. This is called *row major* order.

Chapter 96: Type Inference

This topic discusses about type inferencing that involves the keyword `auto` type that is available from C++11.

Section 96.1: Data Type: Auto

This example shows the basic type inferences the compiler can perform.

```
auto a = 1;        //    a = int
auto b = 2u;       //    b = unsigned int
auto c = &a;       //    c = int*
const auto  d = c; //    d = const int*
const auto& e = b; //    e = const unsigned int&

auto x = a + b     //    x = int, #compiler warning unsigned and signed

auto v = std::vector<int>;    //    v = std::vector<int>
```

However, the auto keyword does not always perform the expected type inference without additional hints for `&` or `const` or `constexpr`

```
//    y = unsigned int,
//    note that y does not infer as const unsigned int&
//    The compiler would have generated a copy instead of a reference value to e or b
auto y = e;
```

Section 96.2: Lambda auto

The data type auto keyword is a convenient way for programmers to declare lambda functions. It helps by shortening the amount of text programmers need to type to declare a function pointer.

```
auto DoThis = [](int a, int b) { return a + b; };
//    Do this is of type (int)(*DoThis)(int, int)
//    else we would have to write this long
int(*pDoThis)(int, int)= [](int a, int b) { return a + b; };

auto c = Dothis(1, 2);    //    c = int
auto d = pDothis(1, 2);   //    d = int

//    using 'auto' shortens the definition for lambda functions
```

By default, if the return type of lambda functions is not defined, it will be automatically inferred from the return expression types.

These 3 is basically the same thing

```
[](int a, int b) -> int  { return a + b; };
[](int a, int b) -> auto { return a + b; };
[](int a, int b) { return a + b; };
```

Section 96.3: Loops and auto

This example shows how auto can be used to shorten type declaration for for loops

```
std::map<int, std::string> Map;
```

```
for (auto pair : Map)            //    pair = std::pair<int, std::string>
for (const auto pair : Map)      //    pair = const std::pair<int, std::string>
for (const auto& pair : Map)     //    pair = const std::pair<int, std::string>&
for (auto i = 0; i < 1000; ++i)         //    i = int
for (auto i = 0; i < Map.size(); ++i)  //    Note that i = int and not size_t
for (auto i = Map.size(); i > 0; --i)  //    i = size_t
```

Chapter 97: Typedef and type aliases

The typedef and (since C++11) using keywords can be used to give a new name to an existing type.

Section 97.1: Basic typedef syntax

A typedef declaration has the same syntax as a variable or function declaration, but it contains the word typedef. The presence of typedef causes the declaration to declare a type instead of a variable or function.

```
int T;         // T has type int
typedef int T; // T is an alias for int

int A[100];         // A has type "array of 100 ints"
typedef int A[100]; // A is an alias for the type "array of 100 ints"
```

Once a type alias has been defined, it can be used interchangeably with the original name of the type.

```
typedef int A[100];
// S is a struct containing an array of 100 ints
struct S {
    A data;
};
```

typedef never creates a distinct type. It only gives another way of referring to an existing type.

```
struct S {
    int f(int);
};
typedef int I;
// ok: defines int S::f(int)
I S::f(I x) { return x; }
```

Section 97.2: More complex uses of typedef

The rule that typedef declarations have the same syntax as ordinary variable and function declarations can be used to read and write more complex declarations.

```
void (*f)(int);         // f has type "pointer to function of int returning void"
typedef void (*f)(int); // f is an alias for "pointer to function of int returning void"
```

This is especially useful for constructs with confusing syntax, such as pointers to non-static members.

```
void (Foo::*pmf)(int);         // pmf has type "pointer to member function of Foo taking int
                               // and returning void"
typedef void (Foo::*pmf)(int); // pmf is an alias for "pointer to member function of Foo
                               // taking int and returning void"
```

It is hard to remember the syntax of the following function declarations, even for experienced programmers:

```
void (Foo::*Foo::f(const char*))(int);
int (&g())[100];
```

typedef can be used to make them easier to read and write:

```
typedef void (Foo::pmf)(int);  // pmf is a pointer to member function type
```

```
pmf Foo::f(const char*);        // f is a member function of Foo

typedef int (&ra)[100];         // ra means "reference to array of 100 ints"
ra g();                         // g returns reference to array of 100 ints
```

Section 97.3: Declaring multiple types with typedef

The `typedef` keyword is a specifier, so it applies separately to each declarator. Therefore, each name declared refers to the type that that name would have in the absence of `typedef`.

```
int *x, (*p)();         // x has type int*, and p has type int(*)()
typedef int *x, (*p)(); // x is an alias for int*, while p is an alias for int(*)()
```

Section 97.4: Alias declaration with "using"

Version ≥ C++11

The syntax of `using` is very simple: the name to be defined goes on the left hand side, and the definition goes on the right hand side. No need to scan to see where the name is.

```
using I = int;
using A = int[100];             // array of 100 ints
using FP = void(*)(int);        // pointer to function of int returning void
using MP = void (Foo::*)(int);  // pointer to member function of Foo of int returning void
```

Creating a type alias with `using` has exactly the same effect as creating a type alias with `typedef`. It is simply an alternative syntax for accomplishing the same thing.

Unlike `typedef`, `using` can be templated. A "template typedef" created with `using` is called an alias template.

Chapter 98: type deduction

Section 98.1: Template parameter deduction for constructors

Prior to C++17, template deduction cannot deduce the class type for you in a constructor. It must be explicitly specified. Sometimes, however, these types can be very cumbersome or (in the case of lambdas) impossible to name, so we got a proliferation of type factories (like `make_pair()`, `make_tuple()`, `back_inserter()`, etc.).

Version ≥ C++17

This is no longer necessary:

```
std::pair p(2, 4.5);     // std::pair<int, double>
std::tuple t(4, 3, 2.5); // std::tuple<int, int, double>
std::copy_n(vi1.begin(), 3,
    std::back_insert_iterator(vi2)); // constructs a back_insert_iterator<std::vector<int>>
std::lock_guard lk(mtx); // std::lock_guard<decltype(mtx)>
```

Constructors are considered to deduce the class template parameters, but in some cases this is insufficient and we can provide explicit deduction guides:

```
template <class Iter>
vector(Iter, Iter) -> vector<typename iterator_traits<Iter>::value_type>

int array[] = {1, 2, 3};
std::vector v(std::begin(array), std::end(array)); // deduces std::vector<int>
```

Section 98.2: Auto Type Deduction

Version ≥ C++11

Type deduction using the `auto` keyword works almost the same as Template Type Deduction. Below are a few examples:

```
auto x = 27;           // (x is neither a pointer nor a reference), x's type is int
const auto cx = x;     // (cx is neither a pointer nor a reference), cs's type is const int
const auto& rx = x;    // (rx is a non-universal reference), rx's type is a reference to a const
int

auto&& uref1 = x;      // x is int and lvalue, so uref1's type is int&
auto&& uref2 = cx;     // cx is const int and lvalue, so uref2's type is const int &
auto&& uref3 = 27;     // 27 is an int and rvalue, so uref3's type is int&&
```

The differences are outlined below:

```
auto x1 = 27;          // type is int, value is 27
auto x2(27);           // type is int, value is 27
auto x3 = { 27 };      // type is std::initializer_list<int>, value is { 27 }
auto x4{ 27 };         // type is std::initializer_list<int>, value is { 27 }
                       // in some compilers type may be deduced as an int with a
                       // value of 27. See remarks for more information.
auto x5 = { 1, 2.0 }   // error! can't deduce T for std::initializer_list<t>
```

As you can see if you use braced initializers, auto is forced into creating a variable of type `std::initializer_list<T>`. If it can't deduce the of T, the code is rejected.

When auto is used as the return type of a function, it specifies that the function has a trailing return type.

```
auto f() -> int {
    return 42;
}
```

Version ≥ C++14

C++14 allows, in addition to the usages of auto allowed in C++11, the following:

1. When used as the return type of a function without a trailing return type, specifies that the function's return type should be deduced from the return statements in the function's body, if any.

   ```
   // f returns int:
   auto f() { return 42; }
   // g returns void:
   auto g() { std::cout << "hello, world!\n"; }
   ```

2. When used in the parameter type of a lambda, defines the lambda to be a generic lambda.

   ```
   auto triple = [](auto x) { return 3*x; };
   const auto x = triple(42); // x is a const int with value 126
   ```

The special form decltype(auto) deduces a type using the type deduction rules of decltype rather than those of auto.

```
int* p = new int(42);
auto x = *p;           // x has type int
decltype(auto) y = *p; // y is a reference to *p
```

In C++03 and earlier, the auto keyword had a completely different meaning as a storage class specifier that was inherited from C.

Section 98.3: Template Type Deduction

Template Generic Syntax

```
template<typename T>
void f(ParamType param);

f(expr);
```

Case 1: ParamType is a Reference or Pointer, but not a Universal or Forward Reference. In this case type deduction works this way. The compiler ignores the reference part if it exists in expr. The compiler then pattern-matches expr's type against ParamType to determing T.

```
template<typename T>
void f(T& param);       //param is a reference

int x = 27;             // x is an int
const int cx = x;       // cx is a const int
const int& rx = x;      // rx is a reference to x as a const int

f(x);                   // T is int, param's type is int&
f(cx);                  // T is const int, param's type is const int&
f(rx);                  // T is const int, param's type is const int&
```

Case 2: `ParamType` is a Universal Reference or Forward Reference. In this case type deduction is the same as in case 1 if the `expr` is an rvalue. If `expr` is an lvalue, both `T` and `ParamType` are deduced to be lvalue references.

```
template<typename T>
void f(T&& param);      // param is a universal reference

int x = 27;             // x is an int
const int cx = x;       // cx is a const int
const int& rx = x;      // rx is a reference to x as a const int

f(x);                   // x is lvalue, so T is int&, param's type is also int&
f(cx);                  // cx is lvalue, so T is const int&, param's type is also const int&
f(rx);                  // rx is lvalue, so T is const int&, param's type is also const int&
f(27);                  // 27 is rvalue, so T is int, param's type is therefore int&&
```

Case 3: `ParamType` is Neither a Pointer nor a Reference. If `expr` is a reference the reference part is ignored. If `expr` is const that is ignored as well. If it is volatile that is also ignored when deducing T's type.

```
template<typename T>
void f(T param);        // param is now passed by value

int x = 27;             // x is an int
const int cx = x;       // cx is a const int
const int& rx = x;      // rx is a reference to x as a const int

f(x);                   // T's and param's types are both int
f(cx);                  // T's and param's types are again both int
f(rx);                  // T's and param's types are still both int
```

Chapter 99: Trailing return type

Section 99.1: Avoid qualifying a nested type name

```
class ClassWithAReallyLongName {
  public:
    class Iterator { /* ... */ };
    Iterator end();
};
```

Defining the member `end` with a trailing return type:

```
auto ClassWithAReallyLongName::end() -> Iterator { return Iterator(); }
```

Defining the member `end` without a trailing return type:

```
ClassWithAReallyLongName::Iterator ClassWithAReallyLongName::end() { return Iterator(); }
```

The trailing return type is looked up in the scope of the class, while a leading return type is looked up in the enclosing namespace scope and can therefore require "redundant" qualification.

Section 99.2: Lambda expressions

A lambda can *only* have a trailing return type; the leading return type syntax is not applicable to lambdas. Note that in many cases it is not necessary to specify a return type for a lambda at all.

```
struct Base {};
struct Derived1 : Base {};
struct Derived2 : Base {};
auto lambda = [](bool b) -> Base* { if (b) return new Derived1; else return new Derived2; };
// ill-formed: auto lambda = Base* [](bool b) { ... };
```

Chapter 100: Alignment

All types in C++ have an alignment. This is a restriction on the memory address that objects of that type can be created within. A memory address is valid for an object's creation if dividing that address by the object's alignment is a whole number.

Type alignments are always a power of two (including 1).

Section 100.1: Controlling alignment

Version ≥ C++11

The `alignas` keyword can be used to force a variable, class data member, declaration or definition of a class, or declaration or definition of an enum, to have a particular alignment, if supported. It comes in two forms:

- `alignas(x)`, where `x` is a constant expression, gives the entity the alignment `x`, if supported.
- `alignas(T)`, where `T` is a type, gives the entity an alignment equal to the alignment requirement of `T`, that is, `alignof(T)`, if supported.

If multiple `alignas` specifiers are applied to the same entity, the strictest one applies.

In this example, the buffer `buf` is guaranteed to be appropriately aligned to hold an `int` object, even though its element type is `unsigned char`, which may have a weaker alignment requirement.

```
alignas(int) unsigned char buf[sizeof(int)];
new (buf) int(42);
```

`alignas` cannot be used to give a type a smaller alignment than the type would have without this declaration:

```
alignas(1) int i; //Il-formed, unless `int` on this platform is aligned to 1 byte.
alignas(char) int j; //Il-formed, unless `int` has the same or smaller alignment than `char`.
```

`alignas`, when given an integer constant expression, must be given a valid alignment. Valid alignments are always powers of two, and must be greater than zero. Compilers are required to support all valid alignments up to the alignment of the type `std::max_align_t`. They *may* support larger alignments than this, but support for allocating memory for such objects is limited. The upper limit on alignments is implementation dependent.

C++17 features direct support in `operator new` for allocating memory for over-aligned types.

Section 100.2: Querying the alignment of a type

Version ≥ c++11

The alignment requirement of a type can be queried using the `alignof` keyword as a unary operator. The result is a constant expression of type `std::size_t`, *i.e.,* it can be evaluated at compile time.

```
#include <iostream>
int main() {
    std::cout << "The alignment requirement of int is: " << alignof(int) << '\n';
}
```

Possible output

> The alignment requirement of int is: 4

If applied to an array, it yields the alignment requirement of the element type. If applied to a reference type, it yields the alignment requirement of the referenced type. (References themselves have no alignment, since they are not objects.)

Chapter 101: Perfect Forwarding

Section 101.1: Factory functions

Suppose we want to write a factory function that accepts an arbitrary list of arguments and passes those arguments unmodified to another function. An example of such a function is `make_unique`, which is used to safely construct a new instance of `T` and return a `unique_ptr<T>` that owns the instance.

The language rules regarding variadic templates and rvalue references allows us to write such a function.

```
template<class T, class... A>
unique_ptr<T> make_unique(A&&... args)
{
    return unique_ptr<T>(new T(std::forward<A>(args)...));
}
```

The use of ellipses `...` indicate a parameter pack, which represents an arbitrary number of types. The compiler will expand this parameter pack to the correct number of arguments at the call site. These arguments are then passed to `T`'s constructor using `std::forward`. This function is required to preserve the ref-qualifiers of the arguments.

```
struct foo
{
    foo() {}
    foo(const foo&) {}                    // copy constructor
    foo(foo&&) {}                         // copy constructor
    foo(int, int, int) {}
};

foo f;
auto p1 = make_unique<foo>(f);            // calls foo::foo(const foo&)
auto p2 = make_unique<foo>(std::move(f)); // calls foo::foo(foo&&)
auto p3 = make_unique<foo>(1, 2, 3);
```

Chapter 102: decltype

The keyword `decltype` can be used to get the type of a variable, function or an expression.

Section 102.1: Basic Example

This example just illustrates how this keyword can be used.

```
int a = 10;

// Assume that type of variable 'a' is not known here, or it may
// be changed by programmer (from int to long long, for example).
// Hence we declare another variable, 'b' of the same type using
// decltype keyword.
decltype(a) b; // 'decltype(a)' evaluates to 'int'
```

If, for example, someone changes, type of 'a' to:

```
float a=99.0f;
```

Then the type of variable `b` now automatically becomes `float`.

Section 102.2: Another example

Let's say we have vector:

```
std::vector<int> intVector;
```

And we want to declare an iterator for this vector. An obvious idea is to use `auto`. However, it may be needed just declare an iterator variable (and not to assign it to anything). We would do:

```
vector<int>::iterator iter;
```

However, with `decltype` it becomes easy and less error prone (if type of `intVector` changes).

```
decltype(intVector)::iterator iter;
```

Alternatively:

```
decltype(intVector.begin()) iter;
```

In second example, the return type of `begin` is used to determine the actual type, which is `vector<int>::iterator`.

If we need a const_iterator, we just need to use `cbegin`:

```
decltype(intVector.cbegin()) iter; // vector<int>::const_iterator
```

Chapter 103: SFINAE (Substitution Failure Is Not An Error)

Section 103.1: What is SFINAE

SFINAE stands for **S**ubstitution **F**ailure **I**s **N**ot **A**n **E**rror. Ill-formed code that results from substituting types (or values) to instantiate a function template or a class template is **not** a hard compile error, it is only treated as a deduction failure.

Deduction failures on instantiating function templates or class template specializations remove that candidate from the set of consideration - as if that failed candidate did not exist to begin with.

```
template <class T>
auto begin(T& c) -> decltype(c.begin()) { return c.begin(); }

template <class T, size_t N>
T* begin(T (&arr)[N]) { return arr; }

int vals[10];
begin(vals); // OK. The first function template substitution fails because
             // vals.begin() is ill-formed. This is not an error! That function
             // is just removed from consideration as a viable overload candidate,
             // leaving us with the array overload.
```

Only substitution failures in the **immediate context** are considered deduction failures, all others are considered hard errors.

```
template <class T>
void add_one(T& val) { val += 1; }

int i = 4;
add_one(i); // ok

std::string msg = "Hello";
add_one(msg); // error. msg += 1 is ill-formed for std::string, but this
              // failure is NOT in the immediate context of substituting T
```

Section 103.2: void_t

Version ≥ C++11

`void_t` is a meta-function that maps any (number of) types to type `void`. The primary purpose of `void_t` is to facilitate writing of type traits.

`std::void_t` will be part of C++17, but until then, it is extremely straightforward to implement:

```
template <class...> using void_t = void;
```

Some compilers require a slightly different implementation:

```
template <class...>
struct make_void { using type = void; };

template <typename... T>
using void_t = typename make_void<T...>::type;
```

The primary application of `void_t` is writing type traits that check validity of a statement. For example, let's check if a type has a member function `foo()` that takes no arguments:

```
template <class T, class=void>
struct has_foo : std::false_type {};

template <class T>
struct has_foo<T, void_t<decltype(std::declval<T&>().foo())>> : std::true_type {};
```

How does this work? When I try to instantiate `has_foo<T>::value`, that will cause the compiler to try to look for the best specialization for `has_foo<T, void>`. We have two options: the primary, and this secondary one which involves having to instantiate that underlying expression:

- If `T` *does* have a member function `foo()`, then whatever type that returns gets converted to `void`, and the specialization is preferred to the primary based on the template partial ordering rules. So `has_foo<T>::value` will be `true`
- If `T` *doesn't* have such a member function (or it requires more than one argument), then substitution fails for the specialization and we only have the primary template to fallback on. Hence, `has_foo<T>::value` is `false`.

A simpler case:

```
template<class T, class=void>
struct can_reference : std::false_type {};

template<class T>
struct can_reference<T, std::void_t<T&>> : std::true_type {};
```

this doesn't use `std::declval` or `decltype`.

You may notice a common pattern of a void argument. We can factor this out:

```
struct details {
  template<template<class...>class Z, class=void, class...Ts>
  struct can_apply:
    std::false_type
  {};
  template<template<class...>class Z, class...Ts>
  struct can_apply<Z, std::void_t<Z<Ts...>>, Ts...>:
    std::true_type
  {};
};

template<template<class...>class Z, class...Ts>
using can_apply = details::can_apply<Z, void, Ts...>;
```

which hides the use of `std::void_t` and makes `can_apply` act like an indicator whether the type supplied as the first template argument is well-formed after substituting the other types into it. The previous examples may now be rewritten using `can_apply` as:

```
template<class T>
using ref_t = T&;

template<class T>
using can_reference = can_apply<ref_t, T>;    // Is T& well formed for T?
```

and:

```
template<class T>
using dot_foo_r = decltype(std::declval<T&>().foo());

template<class T>
using can_dot_foo = can_apply< dot_foo_r, T >;    // Is T.foo() well formed for T?
```

which seems simpler than the original versions.

There are post-C++17 proposals for `std` traits similar to `can_apply`.

The utility of `void_t` was discovered by Walter Brown. He gave a wonderful presentation on it at CppCon 2016.

Section 103.3: enable_if

`std::enable_if` is a convenient utility to use boolean conditions to trigger SFINAE. It is defined as:

```
template <bool Cond, typename Result=void>
struct enable_if { };

template <typename Result>
struct enable_if<true, Result> {
    using type = Result;
};
```

That is, `enable_if<true, R>::type` is an alias for `R`, whereas `enable_if<false, T>::type` is ill-formed as that specialization of `enable_if` does not have a `type` member type.

`std::enable_if` can be used to constrain templates:

```
int negate(int i) { return -i; }

template <class F>
auto negate(F f) { return -f(); }
```

Here, a call to `negate(1)` would fail due to ambiguity. But the second overload is not intended to be used for integral types, so we can add:

```
int negate(int i) { return -i; }

template <class F, class = typename std::enable_if<!std::is_arithmetic<F>::value>::type>
auto negate(F f) { return -f(); }
```

Now, instantiating `negate<int>` would result in a substitution failure since `!std::is_arithmetic<int>::value` is `false`. Due to SFINAE, this is not a hard error, this candidate is simply removed from the overload set. As a result, `negate(1)` only has one single viable candidate - which is then called.

When to use it

It's worth keeping in mind that `std::enable_if` is a helper *on top* of SFINAE, but it's not what makes SFINAE work in the first place. Let's consider these two alternatives for implementing functionality similar to `std::size`, i.e. an overload set `size(arg)` that produces the size of a container or array:

```
// for containers
template<typename Cont>
auto size1(Cont const& cont) -> decltype( cont.size() );
```

```
// for arrays
template<typename Elt, std::size_t Size>
std::size_t size1(Elt const(&arr)[Size]);

// implementation omitted
template<typename Cont>
struct is_sizeable;

// for containers
template<typename Cont, std::enable_if_t<std::is_sizeable<Cont>::value, int> = 0>
auto size2(Cont const& cont);

// for arrays
template<typename Elt, std::size_t Size>
std::size_t size2(Elt const(&arr)[Size]);
```

Assuming that `is_sizeable` is written appropriately, these two declarations should be exactly equivalent with respect to SFINAE. Which is the easiest to write, and which is the easiest to review and understand at a glance?

Now let's consider how we might want to implement arithmetic helpers that avoid signed integer overflow in favour of wrap around or modular behaviour. Which is to say that e.g. `incr(i, 3)` would be the same as `i += 3` save for the fact that the result would always be defined even if `i` is an `int` with value `INT_MAX`. These are two possible alternatives:

```
// handle signed types
template<typename Int>
auto incr1(Int& target, Int amount)
-> std::void_t<int[static_cast<Int>(-1) < static_cast<Int>(0)]>;

// handle unsigned types by just doing target += amount
// since unsigned arithmetic already behaves as intended
template<typename Int>
auto incr1(Int& target, Int amount)
-> std::void_t<int[static_cast<Int>(0) < static_cast<Int>(-1)]>;

template<typename Int, std::enable_if_t<std::is_signed<Int>::value, int> = 0>
void incr2(Int& target, Int amount);

template<typename Int, std::enable_if_t<std::is_unsigned<Int>::value, int> = 0>
void incr2(Int& target, Int amount);
```

Once again which is the easiest to write, and which is the easiest to review and understand at a glance?

A strength of `std::enable_if` is how it plays with refactoring and API design. If `is_sizeable<Cont>::value` is meant to reflect whether `cont.size()` is valid then just using the expression as it appears for `size1` can be more concise, although that could depend on whether `is_sizeable` would be used in several places or not. Contrast that with `std::is_signed` which reflects its intention much more clearly than when its implementation leaks into the declaration of `incr1`.

Section 103.4: is_detected

To generalize type_trait creation:based on SFINAE there are experimental traits `detected_or`, `detected_t`, `is_detected`.

With template parameters `typename Default`, `template <typename...> Op` and `typename ... Args`:

- `is_detected`: alias of `std::true_type` or `std::false_type` depending of the validity of `Op<Args...>`
- `detected_t`: alias of `Op<Args...>` or `nonesuch` depending of validity of `Op<Args...>`.

- `detected_or`: alias of a struct with `value_t` which is `is_detected`, and `type` which is `Op<Args...>` or `Default` depending of validity of `Op<Args...>`

which can be implemented using `std::void_t` for SFINAE as following:

Version ≥ C++17

```
namespace detail {
    template <class Default, class AlwaysVoid,
              template<class...> class Op, class... Args>
    struct detector
    {
        using value_t = std::false_type;
        using type = Default;
    };

    template <class Default, template<class...> class Op, class... Args>
    struct detector<Default, std::void_t<Op<Args...>>, Op, Args...>
    {
        using value_t = std::true_type;
        using type = Op<Args...>;
    };

} // namespace detail

// special type to indicate detection failure
struct nonesuch {
    nonesuch() = delete;
    ~nonesuch() = delete;
    nonesuch(nonesuch const&) = delete;
    void operator=(nonesuch const&) = delete;
};

template <template<class...> class Op, class... Args>
using is_detected =
    typename detail::detector<nonesuch, void, Op, Args...>::value_t;

template <template<class...> class Op, class... Args>
using detected_t = typename detail::detector<nonesuch, void, Op, Args...>::type;

template <class Default, template<class...> class Op, class... Args>
using detected_or = detail::detector<Default, void, Op, Args...>;
```

Traits to detect presence of method can then be simply implemented:

```
typename <typename T, typename ...Ts>
using foo_type = decltype(std::declval<T>().foo(std::declval<Ts>()...));

struct C1 {};

struct C2 {
    int foo(char) const;
};

template <typename T>
using has_foo_char = is_detected<foo_type, T, char>;

static_assert(!has_foo_char<C1>::value, "Unexpected");
static_assert(has_foo_char<C2>::value, "Unexpected");

static_assert(std::is_same<int, detected_t<foo_type, C2, char>>::value,
              "Unexpected");
```

```
static_assert(std::is_same<void, // Default
                           detected_or<void, foo_type, C1, char>>::value,
              "Unexpected");
static_assert(std::is_same<int, detected_or<void, foo_type, C2, char>>::value,
              "Unexpected");
```

Section 103.5: Overload resolution with a large number of options

If you need to select between several options, enabling just one via `enable_if<>` can be quite cumbersome, since several conditions needs to be negated too.

The ordering between overloads can instead be selected using inheritance, i.e. tag dispatch.

Instead of testing for the thing that needs to be well-formed, and also testing the negation of all the other versions conditions, we instead test just for what we need, preferably in a `decltype` in a trailing return.
This might leave several option well formed, we differentiate between those using 'tags', similar to iterator-trait tags (`random_access_tag` et al). This works because a direct match is better that a base class, which is better that a base class of a base class, etc.

```
#include <algorithm>
#include <iterator>

namespace detail
{
    // this gives us infinite types, that inherit from each other
    template<std::size_t N>
    struct pick : pick<N-1> {};
    template<>
    struct pick<0> {};

    // the overload we want to be preferred have a higher N in pick<N>
    // this is the first helper template function
    template<typename T>
    auto stable_sort(T& t, pick<2>)
        -> decltype( t.stable_sort(), void() )
    {
        // if the container have a member stable_sort, use that
        t.stable_sort();
    }

    // this helper will be second best match
    template<typename T>
    auto stable_sort(T& t, pick<1>)
        -> decltype( t.sort(), void() )
    {
        // if the container have a member sort, but no member stable_sort
        // it's customary that the sort member is stable
        t.sort();
    }

    // this helper will be picked last
    template<typename T>
    auto stable_sort(T& t, pick<0>)
        -> decltype( std::stable_sort(std::begin(t), std::end(t)), void() )
    {
        // the container have neither a member sort, nor member stable_sort
        std::stable_sort(std::begin(t), std::end(t));
    }
```

```
}

// this is the function the user calls. it will dispatch the call
// to the correct implementation with the help of 'tags'.
template<typename T>
void stable_sort(T& t)
{
    // use an N that is higher that any used above.
    // this will pick the highest overload that is well formed.
    detail::stable_sort(t, detail::pick<10>{});
}
```

There are other methods commonly used to differentiate between overloads, such as exact match being better than conversion, being better than ellipsis.

However, tag-dispatch can extend to any number of choices, and is a bit more clear in intent.

Section 103.6: trailing decltype in function templates

Version ≥ C++11

One of constraining function is to use trailing `decltype` to specify the return type:

```
namespace details {
   using std::to_string;

   // this one is constrained on being able to call to_string(T)
   template <class T>
   auto convert_to_string(T const& val, int )
       -> decltype(to_string(val))
   {
       return to_string(val);
   }

   // this one is unconstrained, but less preferred due to the ellipsis argument
   template <class T>
   std::string convert_to_string(T const& val, ... )
   {
       std::ostringstream oss;
       oss << val;
       return oss.str();
   }
}

template <class T>
std::string convert_to_string(T const& val)
{
    return details::convert_to_string(val, 0);
}
```

If I call `convert_to_string()` with an argument with which I can invoke `to_string()`, then I have two viable functions for `details::convert_to_string()`. The first is preferred since the conversion from `0` to `int` is a better implicit conversion sequence than the conversion from `0` to `...`

If I call `convert_to_string()` with an argument from which I cannot invoke `to_string()`, then the first function template instantiation leads to substitution failure (there is no `decltype(to_string(val))`). As a result, that candidate is removed from the overload set. The second function template is unconstrained, so it is selected and we instead go through `operator<<(std::ostream&, T)`. If that one is undefined, then we have a hard compile error with a template stack on the line `oss << val`.

Section 103.7: enable_if_all / enable_if_any

Version ≥ C++11

Motivational example

When you have a variadic template pack in the template parameters list, like in the following code snippet:

```
template<typename ...Args> void func(Args &&...args) { //... };
```

The standard library (prior to C++17) offers no direct way to write **enable_if** to impose SFINAE constraints on ***all of the parameters*** in `Args` or ***any of the parameters*** in `Args`. C++17 offers `std::conjunction` and `std::disjunction` which solve this problem. For example:

```
/// C++17: SFINAE constraints on all of the parameters in Args.
template<typename ...Args,
         std::enable_if_t<std::conjunction_v<custom_conditions_v<Args>...>>* = nullptr>
void func(Args &&...args) { //... };

/// C++17: SFINAE constraints on any of the parameters in Args.
template<typename ...Args,
         std::enable_if_t<std::disjunction_v<custom_conditions_v<Args>...>>* = nullptr>
void func(Args &&...args) { //... };
```

If you do not have C++17 available, there are several solutions to achieve these. One of them is to use a base-case class and **partial specializations**, as demonstrated in answers of this question.

Alternatively, one may also implement by hand the behavior of `std::conjunction` and `std::disjunction` in a rather straight-forward way. In the following example I'll demonstrate the implementations and combine them with `std::enable_if` to produce two alias: `enable_if_all` and `enable_if_any`, which do exactly what they are supposed to semantically. This may provide a more scalable solution.

Implementation of `enable_if_all` and `enable_if_any`

First let's emulate `std::conjunction` and `std::disjunction` using customized `seq_and` and `seq_or` respectively:

```
/// Helper for prior to C++14.
template<bool B, class T, class F >
using conditional_t = typename std::conditional<B,T,F>::type;

/// Emulate C++17 std::conjunction.
template<bool...> struct seq_or: std::false_type {};
template<bool...> struct seq_and: std::true_type {};

template<bool B1, bool... Bs>
struct seq_or<B1,Bs...>:
  conditional_t<B1,std::true_type,seq_or<Bs...>> {};

template<bool B1, bool... Bs>
struct seq_and<B1,Bs...>:
  conditional_t<B1,seq_and<Bs...>,std::false_type> {};
```

Then the implementation is quite straight-forward:

```
template<bool... Bs>
using enable_if_any = std::enable_if<seq_or<Bs...>::value>;

template<bool... Bs>
```

```
using enable_if_all = std::enable_if<seq_and<Bs...>::value>;
```

Eventually some helpers:

```
template<bool... Bs>
using enable_if_any_t = typename enable_if_any<Bs...>::type;

template<bool... Bs>
using enable_if_all_t = typename enable_if_all<Bs...>::type;
```

Usage

The usage is also straight-forward:

```
    /// SFINAE constraints on all of the parameters in Args.
    template<typename ...Args,
             enable_if_all_t<custom_conditions_v<Args>...>* = nullptr>
    void func(Args &&...args) { //... };

    /// SFINAE constraints on any of the parameters in Args.
    template<typename ...Args,
             enable_if_any_t<custom_conditions_v<Args>...>* = nullptr>
    void func(Args &&...args) { //... };
```

Chapter 104: Undefined Behavior

What is undefined behavior (UB)? According to the ISO C++ Standard (§1.3.24, N4296), it is "behavior for which this International Standard imposes no requirements."

This means that when a program encounters UB, it is allowed to do whatever it wants. This often means a crash, but it may simply do nothing, make demons fly out of your nose, or even *appear* to work properly!

Needless to say, you should avoid writing code that invokes UB.

Section 104.1: Reading or writing through a null pointer

```
int *ptr = nullptr;
*ptr = 1; // Undefined behavior
```

This is **undefined behavior**, because a null pointer does not point to any valid object, so there is no object at `*ptr` to write to.

Although this most often causes a segmentation fault, it is undefined and anything can happen.

Section 104.2: Using an uninitialized local variable

```
int a;
std::cout << a; // Undefined behavior!
```

This results in **undefined behavior**, because `a` is uninitialised.

It is often, incorrectly, claimed that this is because the value is "indeterminate", or "whatever value was in that memory location before". However, it is the act of accessing the value of `a` in the above example that gives undefined behaviour. In practice, printing a "garbage value" is a common symptom in this case, but that is only one possible form of undefined behaviour.

Although highly unlikely in practice (since it is reliant on specific hardware support) the compiler could equally well electrocute the programmer when compiling the code sample above. With such a compiler and hardware support, such a response to undefined behaviour would markedly increase average (living) programmer understanding of the true meaning of undefined behaviour - which is that the standard places no constraint on the resultant behaviour.

Version ≥ C++14

Using an indeterminate value of `unsigned char` type does not produce undefined behavior if the value is used as:

- the second or third operand of the ternary conditional operator;
- the right operand of the built-in comma operator;
- the operand of a conversion to `unsigned char`;
- the right operand of the assignment operator, if the left operand is also of type `unsigned char`;
- the initializer for an `unsigned char` object;

or if the value is discarded. In such cases, the indeterminate value simply propagates to the result of the expression, if applicable.

Note that a `static` variable is **always** zero-initialized (if possible):

```
static int a;
std::cout << a; // Defined behavior, 'a' is 0
```

Section 104.3: Accessing an out-of-bounds index

It is **undefined behavior** to access an index that is out of bounds for an array (or standard library container for that matter, as they are all implemented using a *raw* array):

```
int array[] = {1, 2, 3, 4, 5};
array[5] = 0;  // Undefined behavior
```

It is *allowed* to have a pointer pointing to the end of the array (in this case `array + 5`), you just can't dereference it, as it is not a valid element.

```
const int *end = array + 5;  // Pointer to one past the last index
for (int *p = array; p != end; ++p)
  // Do something with `p`
```

In general, you're not allowed to create an out-of-bounds pointer. A pointer must point to an element within the array, or one past the end.

Section 104.4: Deleting a derived object via a pointer to a base class that doesn't have a virtual destructor

```
class base { };
class derived: public base { };

int main() {
    base* p = new derived();
    delete p; // The is undefined behavior!
}
```

In section [expr.delete] §5.3.5/3 the standard says that if `delete` is called on an object whose static type does not have a `virtual` destructor:

> if the static type of the object to be deleted is different from its dynamic type, the static type shall be a base class of the dynamic type of the object to be deleted and the static type shall have a virtual destructor or the behavior is undefined.

This is the case regardless of the question whether the derived class added any data members to the base class.

Section 104.5: Extending the `std` or `posix` Namespace

The standard (17.6.4.2.1/1) generally forbids extending the `std` namespace:

> The behavior of a C++ program is undefined if it adds declarations or definitions to namespace std or to a namespace within namespace std unless otherwise specified.

The same goes for `posix` (17.6.4.2.2/1):

> The behavior of a C++ program is undefined if it adds declarations or definitions to namespace posix or to a namespace within namespace posix unless otherwise specified.

Consider the following:

```
#include <algorithm>

namespace std
{
    int foo(){}
}
```

Nothing in the standard forbids `algorithm` (or one of the headers it includes) defining the same definition, and so this code would violate the One Definition Rule.

So, in general, this is forbidden. There are specific exceptions allowed, though. Perhaps most usefully, it is allowed to add specializations for user defined types. So, for example, suppose your code has

```
class foo
{
    // Stuff
};
```

Then the following is fine

```
namespace std
{
    template<>
    struct hash<foo>
    {
    public:
        size_t operator()(const foo &f) const;
    };
}
```

Section 104.6: Invalid pointer arithmetic

The following uses of pointer arithmetic cause undefined behavior:

- Addition or subtraction of an integer, if the result does not belong to the same array object as the pointer operand. (Here, the element one past the end is considered to still belong to the array.)

  ```
  int a[10];
  int* p1 = &a[5];
  int* p2 = p1 + 4; // ok; p2 points to a[9]
  int* p3 = p1 + 5; // ok; p2 points to one past the end of a
  int* p4 = p1 + 6; // UB
  int* p5 = p1 - 5; // ok; p2 points to a[0]
  int* p6 = p1 - 6; // UB
  int* p7 = p3 - 5; // ok; p7 points to a[5]
  ```

- Subtraction of two pointers if they do not both belong to the same array object. (Again, the element one past the end is considered to belong to the array.) The exception is that two null pointers may be subtracted, yielding 0.

```
int a[10];
int b[10];
int *p1 = &a[8], *p2 = &a[3];
int d1 = p1 - p2; // yields 5
int *p3 = p1 + 2; // ok; p3 points to one past the end of a
int d2 = p3 - p2; // yields 7
int *p4 = &b[0];
int d3 = p4 - p1; // UB
```

- Subtraction of two pointers if the result overflows `std::ptrdiff_t`.
- Any pointer arithmetic where either operand's pointee type does not match the dynamic type of the object pointed to (ignoring cv-qualification). According to the standard, "[in] particular, a pointer to a base class cannot be used for pointer arithmetic when the array contains objects of a derived class type."

```
struct Base { int x; };
struct Derived : Base { int y; };
Derived a[10];
Base* p1 = &a[1];           // ok
Base* p2 = p1 + 1;          // UB; p1 points to Derived
Base* p3 = p1 - 1;          // likewise
Base* p4 = &a[2];           // ok
auto p5 = p4 - p1;          // UB; p4 and p1 point to Derived
const Derived* p6 = &a[1];
const Derived* p7 = p6 + 1; // ok; cv-qualifiers don't matter
```

Section 104.7: No return statement for a function with a non-void return type

Omitting the `return` statement in a function which is has a return type that is not `void` is **undefined behavior**.

```
int function() {
    // Missing return statement
}

int main() {
    function(); //Undefined Behavior
}
```

Most modern day compilers emit a warning at compile time for this kind of undefined behavior.

Note: `main` is the only exception to the rule. If `main` doesn't have a `return` statement, the compiler automatically inserts `return 0;` for you, so it can be safely left out.

Section 104.8: Accessing a dangling reference

It is illegal to access a reference to an object that has gone out of scope or been otherwise destroyed. Such a reference is said to be *dangling* since it no longer refers to a valid object.

```
#include <iostream>
int& getX() {
    int x = 42;
    return x;
}
```

```
int main() {
    int& r = getX();
    std::cout << r << "\n";
}
```

In this example, the local variable `x` goes out of scope when `getX` returns. (Note that *lifetime extension* cannot extend the lifetime of a local variable past the scope of the block in which it is defined.) Therefore `r` is a dangling reference. This program has undefined behavior, although it may appear to work and print `42` in some cases.

Section 104.9: Integer division by zero

```
int x = 5 / 0;    // Undefined behavior
```

Division by `0` is mathematically undefined, and as such it makes sense that this is undefined behavior.

However:

```
float x = 5.0f / 0.0f;   // x is +infinity
```

Most implementation implement IEEE-754, which defines floating point division by zero to return `NaN` (if numerator is `0.0f`), `infinity` (if numerator is positive) or `-infinity` (if numerator is negative).

Section 104.10: Shifting by an invalid number of positions

For the built-in shift operator, the right operand must be nonnegative and strictly less than the bit width of the promoted left operand. Otherwise, the behavior is undefined.

```
const int a = 42;
const int b = a << -1; // UB
const int c = a << 0;  // ok
const int d = a << 32; // UB if int is 32 bits or less
const int e = a >> 32; // also UB if int is 32 bits or less
const signed char f = 'x';
const int g = f << 10; // ok even if signed char is 10 bits or less;
                       // int must be at least 16 bits
```

Section 104.11: Incorrect pairing of memory allocation and deallocation

An object can only be deallocated by `delete` if it was allocated by `new` and is not an array. If the argument to `delete` was not returned by `new` or is an array, the behavior is undefined.

An object can only be deallocated by `delete[]` if it was allocated by `new` and is an array. If the argument to `delete[]` was not returned by `new` or is not an array, the behavior is undefined.

If the argument to `free` was not returned by `malloc`, the behavior is undefined.

```
int* p1 = new int;
delete p1;      // correct
// delete[] p1; // undefined
// free(p1);    // undefined

int* p2 = new int[10];
delete[] p2;    // correct
// delete p2;   // undefined
// free(p2);    // undefined
```

```
int* p3 = static_cast<int*>(malloc(sizeof(int)));
free(p3);       // correct
// delete p3;   // undefined
// delete[] p3; // undefined
```

Such issues can be avoided by completely avoiding `malloc` and `free` in C++ programs, preferring the standard library smart pointers over raw `new` and `delete`, and preferring `std::vector` and `std::string` over raw `new` and `delete[]`.

Section 104.12: Signed Integer Overflow

```
int x = INT_MAX + 1;

// x can be anything -> Undefined behavior
```

> If during the evaluation of an expression, the result is not mathematically defined or not in the range of representable values for its type, the behavior is undefined.

(C++11 Standard paragraph 5/4)

This is one of the more nasty ones, as it usually yields reproducible, non-crashing behavior so developers may be tempted to rely heavily on the observed behavior.

On the other hand:

```
unsigned int x = UINT_MAX + 1;

// x is 0
```

is well defined since:

> Unsigned integers, declared unsigned, shall obey the laws of arithmetic modulo `2^n` where `n` is the number of bits in the value representation of that particular size of integer.

(C++11 Standard paragraph 3.9.1/4)

Sometimes compilers may exploit an undefined behavior and optimize

```
signed int x ;
if(x > x + 1)
{
    //do something
}
```

Here since a signed integer overflow is not defined, compiler is free to assume that it may never happen and hence it can optimize away the "if" block

Section 104.13: Multiple non-identical definitions (the One Definition Rule)

If a class, enum, inline function, template, or member of a template has external linkage and is defined in multiple translation units, all definitions must be identical or the behavior is undefined according to the One Definition Rule

(ODR).

foo.h:

```
class Foo {
  public:
    double x;
  private:
    int y;
};

Foo get_foo();
```

foo.cpp:

```
#include "foo.h"
Foo get_foo() { /* implementation */ }
```

main.cpp:

```
// I want access to the private member, so I am going to replace Foo with my own type
class Foo {
  public:
    double x;
    int y;
};
Foo get_foo(); // declare this function ourselves since we aren't including foo.h
int main() {
    Foo foo = get_foo();
    // do something with foo.y
}
```

The above program exhibits undefined behavior because it contains two definitions of the class `::Foo`, which has external linkage, in different translation units, but the two definitions are not identical. Unlike redefinition of a class within the *same* translation unit, this problem is not required to be diagnosed by the compiler.

Section 104.14: Modifying a const object

Any attempt to modify a `const` object results in undefined behavior. This applies to `const` variables, members of `const` objects, and class members declared `const`. (However, a `mutable` member of a `const` object is not `const`.)

Such an attempt can be made through `const_cast`:

```
const int x = 123;
const_cast<int&>(x) = 456;
std::cout << x << '\n';
```

A compiler will usually inline the value of a `const int` object, so it's possible that this code compiles and prints `123`. Compilers can also place `const` objects' values in read-only memory, so a segmentation fault may occur. In any case, the behavior is undefined and the program might do anything.

The following program conceals a far more subtle error:

```
#include <iostream>

class Foo* instance;

```

```
class Foo {
  public:
    int get_x() const { return m_x; }
    void set_x(int x) { m_x = x; }
  private:
    Foo(int x, Foo*& this_ref): m_x(x) {
        this_ref = this;
    }
    int m_x;
    friend const Foo& getFoo();
};

const Foo& getFoo() {
    static const Foo foo(123, instance);
    return foo;
}

void do_evil(int x) {
    instance->set_x(x);
}

int main() {
    const Foo& foo = getFoo();
    do_evil(456);
    std::cout << foo.get_x() << '\n';
}
```

In this code, `getFoo` creates a singleton of type `const Foo` and its member `m_x` is initialized to `123`. Then `do_evil` is called and the value of `foo.m_x` is apparently changed to 456. What went wrong?

Despite its name, `do_evil` does nothing particularly evil; all it does is call a setter through a `Foo*`. But that pointer points to a `const Foo` object even though `const_cast` was not used. This pointer was obtained through `Foo`'s constructor. A `const` object does not become `const` until its initialization is complete, so `this` has type `Foo*`, not `const Foo*`, within the constructor.

Therefore, undefined behavior occurs even though there are no obviously dangerous constructs in this program.

Section 104.15: Returning from a [[noreturn]] function

Version ≥ C++11

Example from the Standard, [dcl.attr.noreturn]:

```
[[ noreturn ]] void f() {
  throw "error"; // OK
}
[[ noreturn ]] void q(int i) { // behavior is undefined if called with an argument <= 0
  if (i > 0)
    throw "positive";
}
```

Section 104.16: Infinite template recursion

Example from the Standard, [temp.inst]/17:

```
template<class T> class X {
    X<T>* p; // OK
    X<T*> a; // implicit generation of X<T> requires
             // the implicit instantiation of X<T*> which requires
```

```
            // the implicit instantiation of X<T**> which ...
};
```

Section 104.17: Overflow during conversion to or from floating point type

If, during the conversion of:

- an integer type to a floating point type,
- a floating point type to an integer type, or
- a floating point type to a shorter floating point type,

the source value is outside the range of values that can be represented in the destination type, the result is undefined behavior. Example:

```
double x = 1e100;
int y = x; // int probably cannot hold numbers that large, so this is UB
```

Section 104.18: Modifying a string literal

Version < C++11

```
char *str = "hello world";
str[0] = 'H';
```

`"hello world"` is a string literal, so modifying it gives undefined behaviour.

The initialisation of `str` in the above example was formally deprecated (scheduled for removal from a future version of the standard) in C++03. A number of compilers before 2003 might issue a warning about this (e.g. a suspicious conversion). After 2003, compilers typically warn about a deprecated conversion.

Version ≥ C++11

The above example is illegal, and results in a compiler diagnostic, in C++11 and later. A similar example may be constructed to exhibit undefined behaviour by explicitly permitting the type conversion, such as:

```
char *str = const_cast<char *>("hello world");
str[0] = 'H';
```

Section 104.19: Accessing an object as the wrong type

In most cases, it is illegal to access an object of one type as though it were a different type (disregarding cv-qualifiers). Example:

```
float x = 42;
int y = reinterpret_cast<int&>(x);
```

The result is undefined behavior.

There are some exceptions to this *strict aliasing* rule:

- An object of class type can be accessed as though it were of a type that is a base class of the actual class type.
- Any type can be accessed as a `char` or `unsigned char`, but the reverse is not true: a char array cannot be accessed as though it were an arbitrary type.
- A signed integer type can be accessed as the corresponding unsigned type and *vice versa*.

A related rule is that if a non-static member function is called on an object that does not actually have the same type as the defining class of the function, or a derived class, then undefined behavior occurs. This is true even if the function does not access the object.

```
struct Base {
};
struct Derived : Base {
    void f() {}
};
struct Unrelated {};
Unrelated u;
Derived& r1 = reinterpret_cast<Derived&>(u); // ok
r1.f();                                      // UB
Base b;
Derived& r2 = reinterpret_cast<Derived&>(b); // ok
r2.f();                                      // UB
```

Section 104.20: Invalid derived-to-base conversion for pointers to members

When `static_cast` is used to convert `T D::*` to `T B::*`, the member pointed to must belong to a class that is a base class or derived class of `B`. Otherwise the behavior is undefined. See Derived to base conversion for pointers to members

Section 104.21: Destroying an object that has already been destroyed

In this example, a destructor is explicitly invoked for an object that will later be automatically destroyed.

```
struct S {
    ~S() { std::cout << "destroying S\n"; }
};
int main() {
    S s;
    s.~S();
} // UB: s destroyed a second time here
```

A similar issue occurs when a `std::unique_ptr<T>` is made to point at a `T` with automatic or static storage duration.

```
void f(std::unique_ptr<S> p);
int main() {
    S s;
    std::unique_ptr<S> p(&s);
    f(std::move(p)); // s destroyed upon return from f
}                    // UB: s destroyed
```

Another way to destroy an object twice is by having two `shared_ptr`s both manage the object without sharing ownership with each other.

```
void f(std::shared_ptr<S> p1, std::shared_ptr<S> p2);
int main() {
    S* p = new S;
    // I want to pass the same object twice...
    std::shared_ptr<S> sp1(p);
    std::shared_ptr<S> sp2(p);
    f(sp1, sp2);
} // UB: both sp1 and sp2 will destroy s separately
```

```
// NB: this is correct:
// std::shared_ptr<S> sp(p);
// f(sp, sp);
```

Section 104.22: Access to nonexistent member through pointer to member

When accessing a non-static member of an object through a pointer to member, if the object does not actually contain the member denoted by the pointer, the behavior is undefined. (Such a pointer to member can be obtained through `static_cast`.)

```
struct Base { int x; };
struct Derived : Base { int y; };
int Derived::*pdy = &Derived::y;
int Base::*pby = static_cast<int Base::*>(pdy);

Base* b1 = new Derived;
b1->*pby = 42; // ok; sets y in Derived object to 42
Base* b2 = new Base;
b2->*pby = 42; // undefined; there is no y member in Base
```

Section 104.23: Invalid base-to-derived static cast

If `static_cast` is used to convert a pointer (resp. reference) to base class to a pointer (resp. reference) to derived class, but the operand does not point (resp. refer) to an object of the derived class type, the behavior is undefined. See Base to derived conversion.

Section 104.24: Floating point overflow

If an arithmetic operation that yields a floating point type produces a value that is not in the range of representable values of the result type, the behavior is undefined according to the C++ standard, but may be defined by other standards the machine might conform to, such as IEEE 754.

```
float x = 1.0;
for (int i = 0; i < 10000; i++) {
    x *= 10.0; // will probably overflow eventually; undefined behavior
}
```

Section 104.25: Calling (Pure) Virtual Members From Constructor Or Destructor

The Standard (10.4) states:

> Member functions can be called from a constructor (or destructor) of an abstract class; the effect of making a virtual call (10.3) to a pure virtual function directly or indirectly for the object being created (or destroyed) from such a constructor (or destructor) is undefined.

More generally, some C++ authorities, e.g. Scott Meyers, suggest never calling virtual functions (even non-pure ones) from constructors and dstructors.

Consider the following example, modified from the above link:

```
class transaction
```

```
{
public:
    transaction(){ log_it(); }
    virtual void log_it() const = 0;
};

class sell_transaction : public transaction
{
public:
    virtual void log_it() const { /* Do something */ }
};
```

Suppose we create a `sell_transaction` object:

```
sell_transaction s;
```

This implicitly calls the constructor of `sell_transaction`, which first calls the constructor of `transaction`. When the constructor of `transaction` is called though, the object is not yet of the type `sell_transaction`, but rather only of the type `transaction`.

Consequently, the call in `transaction::transaction()` to `log_it`, won't do what might seem to be the intuitive thing - namely call `sell_transaction::log_it`.

- If `log_it` is pure virtual, as in this example, the behaviour is undefined.
- If `log_it` is non-pure virtual, `transaction::log_it` will be called.

Section 104.26: Function call through mismatched function pointer type

In order to call a function through a function pointer, the function pointer's type must exactly match the function's type. Otherwise, the behaviour is undefined. Example:

```
int f();
void (*p)() = reinterpret_cast<void(*)()>(f);
p(); // undefined
```

Chapter 105: Overload resolution

Section 105.1: Categorization of argument to parameter cost

Overload resolution partitions the cost of passing an argument to a parameter into one of four different categorizes, called "sequences". Each sequence may include zero, one or several conversions

- Standard conversion sequence

```
void f(int a); f(42);
```

- User defined conversion sequence

```
void f(std::string s); f("hello");
```

- Ellipsis conversion sequence

```
void f(...); f(42);
```

- List initialization sequence

```
void f(std::vector<int> v); f({1, 2, 3});
```

The general principle is that Standard conversion sequences are the cheapest, followed by user defined conversion sequences, followed by ellipsis conversion sequences.

A special case is the list initialization sequence, which does not constitute a conversion (an initializer list is not an expression with a type). Its cost is determined by defining it to be equivalent to one of the other three conversion sequences, depending on the parameter type and form of initializer list.

Section 105.2: Arithmetic promotions and conversions

Converting an integer type to the corresponding promoted type is better than converting it to some other integer type.

```
void f(int x);
void f(short x);
signed char c = 42;
f(c); // calls f(int); promotion to int is better than conversion to short
short s = 42;
f(s); // calls f(short); exact match is better than promotion to int
```

Promoting a `float` to `double` is better than converting it to some other floating point type.

```
void f(double x);
void f(long double x);
f(3.14f); // calls f(double); promotion to double is better than conversion to long double
```

Arithmetic conversions other than promotions are neither better nor worse than each other.

```
void f(float x);
void f(long double x);
f(3.14); // ambiguous
```

```
void g(long x);
void g(long double x);
g(42); // ambiguous
g(3.14); // ambiguous
```

Therefore, in order to ensure that there will be no ambiguity when calling a function `f` with either integral or floating-point arguments of any standard type, a total of eight overloads are needed, so that for each possible argument type, either an overload matches exactly or the unique overload with the promoted argument type will be selected.

```
void f(int x);
void f(unsigned int x);
void f(long x);
void f(unsigned long x);
void f(long long x);
void f(unsigned long long x);
void f(double x);
void f(long double x);
```

Section 105.3: Overloading on Forwarding Reference

You must be very careful when providing a forwarding reference overload as it may match too well:

```
struct A {
    A() = default;           // #1
    A(A const& ) = default;  // #2

    template <class T>
    A(T&& );                 // #3
};
```

The intent here was that `A` is copyable, and that we have this other constructor that might initialize some other member. However:

```
A a;     // calls #1
A b(a);  // calls #3!
```

There are two viable matches for the construction call:

```
A(A const& ); // #2
A(A& );       // #3, with T = A&
```

Both are Exact Matches, but `#3` takes a reference to a less *cv*-qualified object than `#2` does, so it has the better standard conversion sequence and is the best viable function.

The solution here is to always constrain these constructors (e.g. using SFINAE):

```
template <class T,
    class = std::enable_if_t<!std::is_convertible<std::decay_t<T>*, A*>::value>
    >
A(T&& );
```

The type trait here is to exclude any `A` or class publicly and unambiguously derived from `A` from consideration, which would make this constructor ill-formed in the example described earlier (and hence removed from the overload set). As a result, the copy constructor is invoked - which is what we wanted.

Section 105.4: Exact match

An overload without conversions needed for parameter types or only conversions needed between types that are still considered exact matches is preferred over an overload that requires other conversions in order to call.

```
void f(int x);
void f(double x);
f(42); // calls f(int)
```

When an argument binds to a reference to the same type, the match is considered to not require a conversion even if the reference is more cv-qualified.

```
void f(int& x);
void f(double x);
int x = 42;
f(x); // argument type is int; exact match with int&

void g(const int& x);
void g(int x);
g(x); // ambiguous; both overloads give exact match
```

For the purposes of overload resolution, the type "array of `T`" is considered to match exactly with the type "pointer to `T`", and the function type `T` is considered to match exactly with the function pointer type `T*`, even though both require conversions.

```
void f(int* p);
void f(void* p);

void g(int* p);
void g(int (&p)[100]);

int a[100];
f(a); // calls f(int*); exact match with array-to-pointer conversion
g(a); // ambiguous; both overloads give exact match
```

Section 105.5: Overloading on constness and volatility

Passing a pointer argument to a `T*` parameter, if possible, is better than passing it to a `const T*` parameter.

```
struct Base {};
struct Derived : Base {};
void f(Base* pb);
void f(const Base* pb);
void f(const Derived* pd);
void f(bool b);

Base b;
f(&b); // f(Base*) is better than f(const Base*)
Derived d;
f(&d); // f(const Derived*) is better than f(Base*) though;
       // constness is only a "tie-breaker" rule
```

Likewise, passing an argument to a `T&` parameter, if possible, is better than passing it to a `const T&` parameter, even if both have exact match rank.

```
void f(int& r);
void f(const int& r);
```

```
int x;
f(x); // both overloads match exactly, but f(int&) is still better
const int y = 42;
f(y); // only f(const int&) is viable
```

This rule applies to const-qualified member functions as well, where it is important for allowing mutable access to non-const objects and immutable access to const objects.

```
class IntVector {
  public:
    // ...
    int* data() { return m_data; }
    const int* data() const { return m_data; }
  private:
    // ...
    int* m_data;
};
IntVector v1;
int* data1 = v1.data();       // Vector::data() is better than Vector::data() const;
                              // data1 can be used to modify the vector's data
const IntVector v2;
const int* data2 = v2.data(); // only Vector::data() const is viable;
                              // data2 can't be used to modify the vector's data
```

In the same way, a volatile overload will be less preferred than a non-volatile overload.

```
class AtomicInt {
  public:
    // ...
    int load();
    int load() volatile;
  private:
    // ...
};
AtomicInt a1;
a1.load(); // non-volatile overload preferred; no side effect
volatile AtomicInt a2;
a2.load(); // only volatile overload is viable; side effect
static_cast<volatile AtomicInt&>(a1).load(); // force volatile semantics for a1
```

Section 105.6: Name lookup and access checking

Overload resolution occurs *after* name lookup. This means that a better-matching function will not be selected by overload resolution if it loses name lookup:

```
void f(int x);
struct S {
    void f(double x);
    void g() { f(42); } // calls S::f because global f is not visible here,
                        // even though it would be a better match
};
```

Overload resolution occurs *before* access checking. An inaccessible function might be selected by overload resolution if it is a better match than an accessible function.

```
class C {
  public:
    static void f(double x);
```

```
  private:
    static void f(int x);
};
C::f(42); // Error! Calls private C::f(int) even though public C::f(double) is viable.
```

Similarly, overload resolution happens without checking whether the resulting call is well-formed with regards to `explicit`:

```
struct X {
    explicit X(int );
    X(char );
};

void foo(X );
foo({4}); // X(int) is better much, but expression is
          // ill-formed because selected constructor is explicit
```

Section 105.7: Overloading within a class hierarchy

The following examples will use this class hierarchy:

```
struct A { int m; };
struct B : A {};
struct C : B {};
```

The conversion from derived class type to base class type is preferred to user-defined conversions. This applies when passing by value or by reference, as well as when converting pointer-to-derived to pointer-to-base.

```
struct Unrelated {
    Unrelated(B b);
};
void f(A a);
void f(Unrelated u);
B b;
f(b); // calls f(A)
```

A pointer conversion from derived class to base class is also better than conversion to `void*`.

```
void f(A* p);
void f(void* p);
B b;
f(&b); // calls f(A*)
```

If there are multiple overloads within the same chain of inheritance, the most derived base class overload is preferred. This is based on a similar principle as virtual dispatch: the "most specialized" implementation is chosen. However, overload resolution always occurs at compile time and will never implicitly down-cast.

```
void f(const A& a);
void f(const B& b);
C c;
f(c); // calls f(const B&)
B b;
A& r = b;
f(r); // calls f(const A&); the f(const B&) overload is not viable
```

For pointers to members, which are contravariant with respect to the class, a similar rule applies in the opposite direction: the least derived derived class is preferred.

```
void f(int B::*p);
void f(int C::*p);
int A::*p = &A::m;
f(p); // calls f(int B::*)
```

Section 105.8: Steps of Overload Resolution

The steps of overload resolution are:

1. Find candidate functions via name lookup. Unqualified calls will perform both regular unqualified lookup as well as argument-dependent lookup (if applicable).

2. Filter the set of candidate functions to a set of *viable* functions. A viable function for which there exists an implicit conversion sequence between the arguments the function is called with and the parameters the function takes.

```
void f(char);          // (1)
void f(int ) = delete; // (2)
void f();              // (3)
void f(int& );         // (4)

f(4); // 1,2 are viable (even though 2 is deleted!)
      // 3 is not viable because the argument lists don't match
      // 4 is not viable because we cannot bind a temporary to
      //     a non-const lvalue reference
```

3. Pick the best viable candidate. A viable function `F1` is a better function than another viable function `F2` if the implicit conversion sequence for each argument in `F1` is not worse than the corresponding implicit conversion sequence in `F2`, and...:

   3.1. For some argument, the implicit conversion sequence for that argument in `F1` is a better conversion sequence than for that argument in `F2`, or

```
void f(int );  // (1)
void f(char ); // (2)

f(4);  // call (1), better conversion sequence
```

   3.2. In a user-defined conversion, the standard conversion sequence from the return of `F1` to the destination type is a better conversion sequence than that of the return type of `F2`, or

```
struct A
{
    operator int();
    operator double();
} a;

int i = a; // a.operator int() is better than a.operator double() and a conversion
float f = a; // ambiguous
```

   3.3. In a direct reference binding, `F1` has the same kind of reference by `F2` is not, or

```
struct A
{
    operator X&();  // #1
    operator X&&(); // #2
```

```
};
A a;
X& lx = a;  // calls #1
X&& rx = a; // calls #2
```

3.4. F1 is not a function template specialization, but F2 is, or

```
template <class T> void f(T ); // #1
void f(int );                  // #2

f(42); // calls #2, the non-template
```

3.5. F1 and F2 are both function template specializations, but F1 is more specialized than F2.

```
template <class T> void f(T );  // #1
template <class T> void f(T* ); // #2

int* p;
f(p); // calls #2, more specialized
```

The ordering here is significant. The better conversion sequence check happens before the template vs non-template check. This leads to a common error with overloading on forwarding reference:

```
struct A {
    A(A const& ); // #1

    template <class T>
    A(T&& );      // #2, not constrained
};

A a;
A b(a); // calls #2!
        // #1 is not a template but #2 resolves to
        // A(A& ), which is a less cv-qualified reference than #1
        // which makes it a better implicit conversion sequence
```

If there's no single best viable candidate at the end, the call is ambiguous:

```
void f(double ) { }
void f(float ) { }

f(42); // error: ambiguous
```

Chapter 106: Move Semantics

Section 106.1: Move semantics

Move semantics are a way of moving one object to another in C++. For this, we empty the old object and place everything it had in the new object.

For this, we must understand what an rvalue reference is. An rvalue reference (`T&&` where T is the object type) is not much different than a normal reference (`T&`, now called lvalue references). But they act as 2 different types, and so, we can make constructors or functions that take one type or the other, which will be necessary when dealing with move semantics.

The reason why we need two different types is to specify two different behaviors. Lvalue reference constructors are related to copying, while rvalue reference constructors are related to moving.

To move an object, we will use `std::move(obj)`. This function returns an rvalue reference to the object, so that we can steal the data from that object into a new one. There are several ways of doing this which are discussed below.

Important to note is that the use of `std::move` creates just an rvalue reference. In other words the statement `std::move(obj)` does not change the content of obj, while `auto obj2 = std::move(obj)` (possibly) does.

Section 106.2: Using std::move to reduce complexity from O(n²) to O(n)

C++11 introduced core language and standard library support for **moving** an object. The idea is that when an object *o* is a temporary and one wants a logical copy, then its safe to just pilfer *o*'s resources, such as a dynamically allocated buffer, leaving *o* logically empty but still destructible and copyable.

The core language support is mainly

- the **rvalue reference** type builder `&&`, e.g., `std::string&&` is an rvalue reference to a `std::string`, indicating that that referred to object is a temporary whose resources can just be pilfered (i.e. moved)
- special support for a **move constructor** `T( T&& )`, which is supposed to efficiently move resources from the specified other object, instead of actually copying those resources, and
- special support for a **move assignment operator** `auto operator=(T&&) -> T&`, which also is supposed to move from the source.

The standard library support is mainly the `std::move` function template from the **`<utility>`** header. This function produces an rvalue reference to the specified object, indicating that it can be moved from, just as if it were a temporary.

For a container actual copying is typically of O(*n*) complexity, where *n* is the number of items in the container, while moving is O(1), constant time. And for an algorithm that logically copies that container *n* times, this can reduce the complexity from the usually impractical O(n^2) to just linear O(*n*).

In his article "Containers That Never Change" in Dr. Dobbs Journal in September 19 2013, Andrew Koenig presented an interesting example of algorithmic inefficiency when using a style of programming where variables are immutable after initialization. With this style loops are generally expressed using recursion. And for some algorithms such as generating a Collatz sequence, the recursion requires logically copying a container:

```
// Based on an example by Andrew Koenig in his Dr. Dobbs Journal article
```

```
// "Containers That Never Change" September 19, 2013, available at
// <url: http://www.drdobbs.com/cpp/containters-that-never-change/240161543>

// Includes here, e.g. <vector>

namespace my {
    template< class Item >
    using Vector_ = /* E.g. std::vector<Item> */;

    auto concat( Vector_<int> const& v, int const x )
        -> Vector_<int>
    {
        auto result{ v };
        result.push_back( x );
        return result;
    }

    auto collatz_aux( int const n, Vector_<int> const& result )
        -> Vector_<int>
    {
        if( n == 1 )
        {
            return result;
        }
        auto const new_result = concat( result, n );
        if( n % 2 == 0 )
        {
            return collatz_aux( n/2, new_result );
        }
        else
        {
            return collatz_aux( 3*n + 1, new_result );
        }
    }

    auto collatz( int const n )
        -> Vector_<int>
    {
        assert( n != 0 );
        return collatz_aux( n, Vector_<int>() );
    }
}  // namespace my

#include <iostream>
using namespace std;
auto main() -> int
{
    for( int const x : my::collatz( 42 ) )
    {
        cout << x << ' ';
    }
    cout << '\n';
}
```

Output:

42 21 64 32 16 8 4 2

The number of item copy operations due to copying of the vectors is here roughly O(n^2), since it's the sum 1 + 2 + 3 + ... *n*.

In concrete numbers, with g++ and Visual C++ compilers the above invocation of `collatz(42)` resulted in a Collatz sequence of 8 items and 36 item copy operations (8*7/2 = 28, plus some) in vector copy constructor calls.

All of these item copy operations can be removed by simply moving vectors whose values are not needed anymore. To do this it's necessary to remove `const` and reference for the vector type arguments, passing the vectors *by value*. The function returns are already automatically optimized. For the calls where vectors are passed, and not used again further on in the function, just apply `std::move` to *move* those buffers rather than actually copying them:

```
using std::move;

auto concat( Vector_<int> v, int const x )
    -> Vector_<int>
{
    v.push_back( x );
    // warning: moving a local object in a return statement prevents copy elision [-Wpessimizing-
move]
    // See https://stackoverflow.com/documentation/c%2b%2b/2489/copy-elision
    // return move( v );
    return v;
}

auto collatz_aux( int const n, Vector_<int> result )
    -> Vector_<int>
{
    if( n == 1 )
    {
        return result;
    }
    auto new_result = concat( move( result ), n );
    struct result;      // Make absolutely sure no use of `result` after this.
    if( n % 2 == 0 )
    {
        return collatz_aux( n/2, move( new_result ) );
    }
    else
    {
        return collatz_aux( 3*n + 1, move( new_result ) );
    }
}

auto collatz( int const n )
    -> Vector_<int>
{
    assert( n != 0 );
    return collatz_aux( n, Vector_<int>() );
}
```

Here, with g++ and Visual C++ compilers, the number of item copy operations due to vector copy constructor invocations, was exactly 0.

The algorithm is necessarily still O(n) in the length of the Collatz sequence produced, but this is a quite dramatic improvement: O(n^2) → O(n).

With some language support one could perhaps use moving and still express and enforce the immutability of a variable *between its initialization and final move*, after which any use of that variable should be an error. Alas, as of C++14 C++ does not support that. For loop-free code the no use after move can be enforced via a re-declaration of the relevant name as an incomplete `struct`, as with `struct result;` above, but this is ugly and not likely to be understood by other programmers; also the diagnostics can be quite misleading.

Summing up, the C++ language and library support for moving allows drastic improvements in algorithm complexity, but due the support's incompleteness, at the cost of forsaking the code correctness guarantees and code clarity that `const` can provide.

For completeness, the instrumented vector class used to measure the number of item copy operations due to copy constructor invocations:

```
template< class Item >
class Copy_tracking_vector
{
private:
    static auto n_copy_ops()
        -> int&
    {
        static int value;
        return value;
    }

    vector<Item>    items_;

public:
    static auto n() -> int { return n_copy_ops(); }

    void push_back( Item const& o ) { items_.push_back( o ); }
    auto begin() const { return items_.begin(); }
    auto end() const { return items_.end(); }

    Copy_tracking_vector(){}

    Copy_tracking_vector( Copy_tracking_vector const& other )
        : items_( other.items_ )
    { n_copy_ops() += items_.size(); }

    Copy_tracking_vector( Copy_tracking_vector&& other )
        : items_( move( other.items_ ) )
    {}
};
```

Section 106.3: Move constructor

Say we have this code snippet.

```
class A {
public:
    int a;
    int b;

    A(const A &other) {
        this->a = other.a;
        this->b = other.b;
    }
};
```

To create a copy constructor, that is, to make a function that copies an object and creates a new one, we normally would choose the syntax shown above, we would have a constructor for A that takes an reference to another object of type A, and we would copy the object manually inside the method.

Alternatively, we could have written `A(const A &) = default;` which automatically copies over all members,

making use of its copy constructor.

To create a move constructor, however, we will be taking an rvalue reference instead of an lvalue reference, like here.

```
class Wallet {
public:
    int nrOfDollars;

    Wallet() = default; //default ctor

    Wallet(Wallet &&other) {
        this->nrOfDollars = other.nrOfDollars;
        other.nrOfDollars = 0;
    }
};
```

Please notice that we set the old values to `zero`. The default move constructor (`Wallet(Wallet&&) = default;`) copies the value of `nrOfDollars`, as it is a POD.

As move semantics are designed to allow 'stealing' state from the original instance, it is important to consider how the original instance should look like after this stealing. In this case, if we would not change the value to zero we would have doubled the amount of dollars into play.

```
Wallet a;
a.nrOfDollars = 1;
Wallet b (std::move(a)); //calling B(B&& other);
std::cout << a.nrOfDollars << std::endl; //0
std::cout << b.nrOfDollars << std::endl; //1
```

Thus we have move constructed an object from an old one.

While the above is a simple example, it shows what the move constructor is intended to do. It becomes more useful in more complex cases, such as when resource management is involved.

```
    // Manages operations involving a specified type.
    // Owns a helper on the heap, and one in its memory (presumably on the stack).
    // Both helpers are DefaultConstructible, CopyConstructible, and MoveConstructible.
    template<typename T,
             template<typename> typename HeapHelper,
             template<typename> typename StackHelper>
    class OperationsManager {
        using MyType = OperationsManager<T, HeapHelper, StackHelper>;

        HeapHelper<T>* h_helper;
        StackHelper<T> s_helper;
        // ...

      public:
        // Default constructor & Rule of Five.
        OperationsManager() : h_helper(new HeapHelper<T>) {}
        OperationsManager(const MyType& other)
          : h_helper(new HeapHelper<T>(*other.h_helper)), s_helper(other.s_helper) {}
        MyType& operator=(MyType copy) {
            swap(*this, copy);
            return *this;
        }
        ~OperationsManager() {
            if (h_helper) { delete h_helper; }
```

```
        }

        // Move constructor (without swap()).
        // Takes other's HeapHelper<T>*.
        // Takes other's StackHelper<T>, by forcing the use of StackHelper<T>'s move constructor.
        // Replaces other's HeapHelper<T>* with nullptr, to keep other from deleting our shiny
        //  new helper when it's destroyed.
        OperationsManager(MyType&& other) noexcept
          : h_helper(other.h_helper),
            s_helper(std::move(other.s_helper)) {
            other.h_helper = nullptr;
        }

        // Move constructor (with swap()).
        // Places our members in the condition we want other's to be in, then switches members
        //  with other.
        // OperationsManager(MyType&& other) noexcept : h_helper(nullptr) {
        //     swap(*this, other);
        // }

        // Copy/move helper.
        friend void swap(MyType& left, MyType& right) noexcept {
            std::swap(left.h_helper, right.h_helper);
            std::swap(left.s_helper, right.s_helper);
        }
    };
```

Section 106.4: Re-use a moved object

You can re-use a moved object:

```
void consumingFunction(std::vector<int> vec) {
    // Some operations
}

int main() {
    // initialize vec with 1, 2, 3, 4
    std::vector<int> vec{1, 2, 3, 4};

    // Send the vector by move
    consumingFunction(std::move(vec));

    // Here the vec object is in an indeterminate state.
    // Since the object is not destroyed, we can assign it a new content.
    // We will, in this case, assign an empty value to the vector,
    // making it effectively empty
    vec = {};

    // Since the vector as gained a determinate value, we can use it normally.
    vec.push_back(42);

    // Send the vector by move again.
    consumingFunction(std::move(vec));
}
```

Section 106.5: Move assignment

Similarly to how we can assign a value to an object with an lvalue reference, copying it, we can also move the values from an object to another without constructing a new one. We call this move assignment. We move the values from

one object to another existing object.

For this, we will have to overload `operator =`, not so that it takes an lvalue reference, like in copy assignment, but so that it takes an rvalue reference.

```
class A {
    int a;
    A& operator= (A&& other) {
        this->a = other.a;
        other.a = 0;
        return *this;
    }
};
```

This is the typical syntax to define move assignment. We overload `operator =` so that we can feed it an rvalue reference and it can assign it to another object.

```
A a;
a.a = 1;
A b;
b = std::move(a); //calling A& operator= (A&& other)
std::cout << a.a << std::endl; //0
std::cout << b.a << std::endl; //1
```

Thus, we can move assign an object to another one.

Section 106.6: Using move semantics on containers

You can move a container instead of copying it:

```
void print(const std::vector<int>& vec) {
    for (auto&& val : vec) {
        std::cout << val << ", ";
    }
    std::cout << std::endl;
}

int main() {
    // initialize vec1 with 1, 2, 3, 4 and vec2 as an empty vector
    std::vector<int> vec1{1, 2, 3, 4};
    std::vector<int> vec2;

    // The following line will print 1, 2, 3, 4
    print(vec1);

    // The following line will print a new line
    print(vec2);

    // The vector vec2 is assigned with move assignment.
    // This will "steal" the value of vec1 without copying it.
    vec2 = std::move(vec1);

    // Here the vec1 object is in an indeterminate state, but still valid.
    // The object vec1 is not destroyed,
    // but there's is no guarantees about what it contains.

    // The following line will print 1, 2, 3, 4
    print(vec2);
```

```
}
```

Chapter 107: Pimpl Idiom

Section 107.1: Basic Pimpl idiom

Version ≥ C++11

In the header file:

```
// widget.h

#include <memory>  // std::unique_ptr
#include <experimental/propagate_const>

class Widget
{
    public:
        Widget();
        ~Widget();
        void DoSomething();

    private:
        // the pImpl idiom is named after the typical variable name used
        // ie, pImpl:
        struct Impl;                    // forward declaration
        std::experimental::propagate_const<std::unique_ptr< Impl >> pImpl;  // ptr to actual
implementation
};
```

In the implementation file:

```
// widget.cpp

#include "widget.h"
#include "reallycomplextype.h" // no need to include this header inside widget.h

struct Widget::Impl
{
    // the attributes needed from Widget go here
    ReallyComplexType rct;
};

Widget::Widget() :
    pImpl(std::make_unique<Impl>())
{}

Widget::~Widget() = default;

void Widget::DoSomething()
{
    // do the stuff here with pImpl
}
```

The `pImpl` contains the `Widget` state (or some/most of it). Instead of the `Widget` description of state being exposed in the header file, it can be only exposed within the implementation.

`pImpl` stands for "pointer to implementation". The "real" implementation of `Widget` is in the `pImpl`.

Danger: Note that for this to work with `unique_ptr`, `~Widget()` must be implemented at a point in a file where the

`Impl` is fully visible. You can `=default` it there, but if `=default` where `Impl` is undefined, the program may easily become ill-formed, no diagnostic required.

Chapter 108: auto

Section 108.1: Basic auto sample

The keyword `auto` provides the auto-deduction of type of a variable.

It is especially convenient when dealing with long type names:

```
std::map< std::string, std::shared_ptr< Widget > > table;
// C++98
std::map< std::string, std::shared_ptr< Widget > >::iterator i = table.find( "42" );
// C++11/14/17
auto j = table.find( "42" );
```

with range-based for loops:

```
vector<int> v = {0, 1, 2, 3, 4, 5};
for(auto n: v)
    std::cout << n << ' ';
```

with lambdas:

```
auto f = [](){ std::cout << "lambda\n"; };
f();
```

to avoid the repetition of the type:

```
auto w = std::make_shared< Widget >();
```

to avoid surprising and unnecessary copies:

```
auto myMap = std::map<int,float>();
myMap.emplace(1,3.14);

std::pair<int,float> const& firstPair2 = *myMap.begin();  // copy!
auto const& firstPair = *myMap.begin();  // no copy!
```

The reason for the copy is that the returned type is actually `std::pair<const int,float>`!

Section 108.2: Generic lambda (C++14)

Version ≥ C++14

C++14 allows to use `auto` in lambda argument

```
auto print = [](const auto& arg) { std::cout << arg << std::endl; };

print(42);
print("hello world");
```

That lambda is mostly equivalent to

```
struct lambda {
    template <typename T>
    auto operator ()(const T& arg) const {
```

```
        std::cout << arg << std::endl;
    }
};
```

and then

```
lambda print;

print(42);
print("hello world");
```

Section 108.3: auto and proxy objects

Sometimes `auto` may behave not quite as was expected by a programmer. It type deduces the expression, even when type deduction is not the right thing to do.

As an example, when proxy objects are used in the code:

```
std::vector<bool> flags{true, true, false};
auto flag = flags[0];
flags.push_back(true);
```

Here `flag` would be not `bool`, but `std::vector<bool>::reference`, since for `bool` specialization of template `vector` the `operator []` returns a proxy object with conversion operator `operator bool` defined.

When `flags.push_back(true)` modifies the container, this pseudo-reference could end up dangling, referring to an element that no longer exists.

It also makes the next situation possible:

```
void foo(bool b);

std::vector<bool> getFlags();

auto flag = getFlags()[5];
foo(flag);
```

The `vector` is discarded immediately, so `flag` is a pseudo-reference to an element that has been discarded. The call to `foo` invokes undefined behavior.

In cases like this you can declare a variable with `auto` and initialize it by casting to the type you want to be deduced:

```
auto flag = static_cast<bool>(getFlags()[5]);
```

but at that point, simply replacing `auto` with `bool` makes more sense.

Another case where proxy objects can cause problems is expression templates. In that case, the templates are sometimes not designed to last beyond the current full-expression for efficiency sake, and using the proxy object on the next causes undefined behavior.

Section 108.4: auto and Expression Templates

`auto` can also cause problems where expression templates come into play:

```
auto mult(int c) {
    return c * std::valarray<int>{1};
```

```
}

auto v = mult(3);
std::cout << v[0]; // some value that could be, but almost certainly is not, 3.
```

The reason is that `operator*` on `valarray` gives you a proxy object that refers to the `valarray` as a means of lazy evaluation. By using `auto`, you're creating a dangling reference. Instead of `mult` had returned a `std::valarray<int>`, then the code would definitely print 3.

Section 108.5: auto, const, and references

The `auto` keyword by itself represents a value type, similar to `int` or `char`. It can be modified with the `const` keyword and the `&` symbol to represent a const type or a reference type, respectively. These modifiers can be combined.

In this example, `s` is a value type (its type will be inferred as `std::string`), so each iteration of the `for` loop *copies* a string from the vector into `s`.

```
std::vector<std::string> strings = { "stuff", "things", "misc" };
for(auto s : strings) {
    std::cout << s << std::endl;
}
```

If the body of the loop modifies `s` (such as by calling `s.append(" and stuff")`), only this copy will be modified, not the original member of `strings`.

On the other hand, if `s` is declared with `auto&` it will be a reference type (inferred to be `std::string&`), so on each iteration of the loop it will be assigned a *reference* to a string in the vector:

```
for(auto& s : strings) {
    std::cout << s << std::endl;
}
```

In the body of this loop, modifications to `s` will directly affect the element of `strings` that it references.

Finally, if `s` is declared `const auto&`, it will be a const reference type, meaning that on each iteration of the loop it will be assigned a *const reference* to a string in the vector:

```
for(const auto& s : strings) {
    std::cout << s << std::endl;
}
```

Within the body of this loop, `s` cannot be modified (i.e. no non-const methods can be called on it).

When using `auto` with range-based `for` loops, it is generally good practice to use `const auto&` if the loop body will not modify the structure being looped over, since this avoids unnecessary copies.

Section 108.6: Trailing return type

`auto` is used in the syntax for trailing return type:

```
auto main() -> int {}
```

which is equivalent to

```
int main() {}
```

Mostly useful combined with `decltype` to use parameters instead of `std::declval<T>`:

```
template <typename T1, typename T2>
auto Add(const T1& lhs, const T2& rhs) -> decltype(lhs + rhs) { return lhs + rhs; }
```

Chapter 109: Copy Elision

Section 109.1: Purpose of copy elision

There are places in the standard where an object is copied or moved in order to initialize an object. Copy elision (sometimes called return value optimization) is an optimization whereby, under certain specific circumstances, a compiler is permitted to avoid the copy or move even though the standard says that it must happen.

Consider the following function:

```
std::string get_string()
{
  return std::string("I am a string.");
}
```

According to the strict wording of the standard, this function will initialize a temporary `std::string`, then copy/move that into the return value object, then destroy the temporary. The standard is very clear that this is how the code is interpreted.

Copy elision is a rule that permits a C++ compiler to *ignore* the creation of the temporary and its subsequent copy/destruction. That is, the compiler can take the initializing expression for the temporary and initialize the function's return value from it directly. This obviously saves performance.

However, it does have two visible effects on the user:

1. The type must have the copy/move constructor that would have been called. Even if the compiler elides the copy/move, the type must still be able to have been copied/moved.
2. Side-effects of copy/move constructors are not guaranteed in circumstances where elision can happen. Consider the following:

Version ≥ C++11

```
struct my_type
{
  my_type() = default;
  my_type(const my_type &) {std::cout <<"Copying\n";}
  my_type(my_type &&) {std::cout <<"Moving\n";}
};

my_type func()
{
  return my_type();
}
```

What will calling `func` do? Well, it will never print "Copying", since the temporary is an rvalue and `my_type` is a moveable type. So will it print "Moving"?

Without the copy elision rule, this would be required to always print "Moving". But because the copy elision rule exists, the move constructor may or may not be called; it is implementation-dependent.

And therefore, you cannot depend on the calling of copy/move constructors in contexts where elision is possible.

Because elision is an optimization, your compiler may not support elision in all cases. And regardless of whether the compiler elides a particular case or not, the type must still support the operation being elided. So if a copy

construction is elided, the type must still have a copy constructor, even though it will not be called.

Section 109.2: Guaranteed copy elision

Version ≥ C++17

Normally, elision is an optimization. While virtually every compiler support copy elision in the simplest of cases, having elision still places a particular burden on users. Namely, the type who's copy/move is being elided *must* still have the copy/move operation that was elided.

For example:

```
std::mutex a_mutex;
std::lock_guard<std::mutex> get_lock()
{
  return std::lock_guard<std::mutex>(a_mutex);
}
```

This might be useful in cases where `a_mutex` is a mutex that is privately held by some system, yet an external user might want to have a scoped lock to it.

This is also not legal, because `std::lock_guard` cannot be copied or moved. Even though virtually every C++ compiler will elide the copy/move, the standard still *requires* the type to have that operation available.

Until C++17.

C++17 mandates elision by effectively redefining the very meaning of certain expressions so that no copy/moving takes place. Consider the above code.

Under pre-C++17 wording, that code says to create a temporary and then use the temporary to copy/move into the return value, but the temporary copy can be elided. Under C++17 wording, that does not create a temporary at all.

In C++17, any prvalue expression, when used to initialize an object of the same type as the expression, does not generate a temporary. The expression directly initializes that object. If you return a prvalue of the same type as the return value, then the type need not have a copy/move constructor. And therefore, under C++17 rules, the above code can work.

The C++17 wording works in cases where the prvalue's type matches the type being initialized. So given `get_lock` above, this will also not require a copy/move:

```
std::lock_guard the_lock = get_lock();
```

Since the result of `get_lock` is a prvalue expression being used to initialize an object of the same type, no copying or moving will happen. That expression never creates a temporary; it is used to directly initialize `the_lock`. There is no elision because there is no copy/move to be elided elide.

The term "guaranteed copy elision" is therefore something of a misnomer, but that is the name of the feature as it is proposed for C++17 standardization. It does not guarantee elision at all; it *eliminates* the copy/move altogether, redefining C++ so that there never was a copy/move to be elided.

This feature only works in cases involving a prvalue expression. As such, this uses the usual elision rules:

```
std::mutex a_mutex;
std::lock_guard<std::mutex> get_lock()
{
  std::lock_guard<std::mutex> my_lock(a_mutex);
```

```
  //Do stuff
  return my_lock;
}
```

While this is a valid case for copy elision, C++17 rules do not *eliminate* the copy/move in this case. As such, the type must still have a copy/move constructor to use to initialize the return value. And since `lock_guard` does not, this is still a compile error. Implementations are allowed to refuse to elide copies when passing or returning an object of trivially-copyable type. This is to allow moving such objects around in registers, which some ABIs might mandate in their calling conventions.

```
struct trivially_copyable {
    int a;
};

void foo (trivially_copyable a) {}

foo(trivially_copyable{}); //copy elision not mandated
```

Section 109.3: Parameter elision

When you pass an argument to a function, and the argument is a prvalue expression of the function's parameter type, and this type is not a reference, then the prvalue's construction can be elided.

```
void func(std::string str) { ... }

func(std::string("foo"));
```

This says to create a temporary `string`, then move it into the function parameter `str`. Copy elision permits this expression to directly create the object in `str`, rather than using a temporary+move.

This is a useful optimization for cases where a constructor is declared `explicit`. For example, we could have written the above as `func("foo")`, but only because `string` has an implicit constructor that converts from a `const char*` to a `string`. If that constructor was `explicit`, we would be forced to use a temporary to call the `explicit` constructor. Copy elision saves us from having to do a needless copy/move.

Section 109.4: Return value elision

If you return a prvalue expression from a function, and the prvalue expression has the same type as the function's return type, then the copy from the prvalue temporary can be elided:

```
std::string func()
{
  return std::string("foo");
}
```

Pretty much all compilers will elide the temporary construction in this case.

Section 109.5: Named return value elision

If you return an lvalue expression from a function, and this lvalue:

- represents an automatic variable local to that function, which will be destroyed after the `return`
- the automatic variable is not a function parameter
- and the type of the variable is the same type as the function's return type

If all of these are the case, then the copy/move from the lvalue can be elided:

```
std::string func()
{
  std::string str("foo");
  //Do stuff
  return str;
}
```

More complex cases are eligible for elision, but the more complex the case, the less likely the compiler will be to actually elide it:

```
std::string func()
{
  std::string ret("foo");
  if(some_condition)
  {
    return "bar";
  }
  return ret;
}
```

The compiler could still elide `ret`, but the chances of them doing so go down.

As noted earlier, elision is not permitted for value *parameters*.

```
std::string func(std::string str)
{
  str.assign("foo");
  //Do stuff
  return str; //No elision possible
}
```

Section 109.6: Copy initialization elision

If you use a prvalue expression to copy initialize a variable, and that variable has the same type as the prvalue expression, then the copying can be elided.

```
std::string str = std::string("foo");
```

Copy initialization effectively transforms this into `std::string str("foo");` (there are minor differences).

This also works with return values:

```
std::string func()
{
  return std::string("foo");
}

std::string str = func();
```

Without copy elision, this would provoke 2 calls to `std::string`'s move constructor. Copy elision permits this to call the move constructor 1 or zero times, and most compilers will opt for the latter.

Chapter 110: Fold Expressions

Section 110.1: Unary Folds

Unary folds are used to *fold* parameter packs over a specific operator. There are 2 kinds of unary folds:

- Unary **Left** Fold `(... op pack)` which expands as follows:

  ```
  ((Pack1 op Pack2) op ...) op PackN
  ```

- Unary **Right** Fold `(pack op ...)` which expands as follows:

  ```
  Pack1 op (... (Pack(N-1) op PackN))
  ```

Here is an example

```
template<typename... Ts>
int sum(Ts... args)
{
    return (... + args); //Unary left fold
    //return (args + ...); //Unary right fold

    // The two are equivalent if the operator is associative.
    // For +, ((1+2)+3) (left fold) == (1+(2+3)) (right fold)
    // For -, ((1-2)-3) (left fold) != (1-(2-3)) (right fold)
}

int result = sum(1, 2, 3); //  6
```

Section 110.2: Binary Folds

Binary folds are basically unary folds, with an extra argument.

There are 2 kinds of binary folds:

- Binary **Left** Fold - `(value op ... op pack)` - Expands as follows:

  ```
  (((Value op Pack1) op Pack2) op ...) op PackN
  ```

- Binary **Right** Fold `(pack op ... op value)` - Expands as follows:

  ```
  Pack1 op (... op (Pack(N-1) op (PackN op Value)))
  ```

Here is an example:

```
template<typename... Ts>
int removeFrom(int num, Ts... args)
{
    return (num - ... - args); //Binary left fold
    // Note that a binary right fold cannot be used
    // due to the lack of associativity of operator-
}
```

```
int result = removeFrom(1000, 5, 10, 15); //'result' is 1000 - 5 - 10 - 15 = 970
```

Section 110.3: Folding over a comma

It is a common operation to need to perform a particular function over each element in a parameter pack. With C++11, the best we can do is:

```
template <class... Ts>
void print_all(std::ostream& os, Ts const&... args) {
    using expander = int[];
    (void)expander{0,
        (void(os << args), 0)...
    };
}
```

But with a fold expression, the above simplifies nicely to:

```
template <class... Ts>
void print_all(std::ostream& os, Ts const&... args) {
    (void(os << args), ...);
}
```

No cryptic boilerplate required.

Chapter 111: Unions

Section 111.1: Undefined Behavior

```
union U {
    int a;
    short b;
    float c;
};
U u;

u.a = 10;
if (u.b == 10) {
   // this is undefined behavior since 'a' was the last member to be
   // written to. A lot of compilers will allow this and might issue a
   // warning, but the result will be "as expected"; this is a compiler
   // extension and cannot be guaranteed across compilers (i.e. this is
   // not compliant/portable code).
}
```

Section 111.2: Basic Union Features

Unions are a specialized struct within which all members occupy overlapping memory.

```
union U {
    int a;
    short b;
    float c;
};
U u;

//Address of a and b will be equal
(void*)&u.a == (void*)&u.b;
(void*)&u.a == (void*)&u.c;

//Assigning to any union member changes the shared memory of all members
u.c = 4.f;
u.a = 5;
u.c != 4.f;
```

Section 111.3: Typical Use

Unions are useful for minimizing memory usage for exclusive data, such as when implementing mixed data types.

```
struct AnyType {
    enum {
        IS_INT,
        IS_FLOAT
    } type;

    union Data {
        int as_int;
        float as_float;
    } value;

    AnyType(int i) : type(IS_INT) { value.as_int = i; }
    AnyType(float f) : type(IS_FLOAT) { value.as_float = f; }
```

```
    int get_int() const {
        if(type == IS_INT)
            return value.as_int;
        else
            return (int)value.as_float;
    }

    float get_float() const {
        if(type == IS_FLOAT)
            return value.as_float;
        else
            return (float)value.as_int;
    }
};
```

Chapter 112: Design pattern implementation in C++

On this page, you can find examples of how design patterns are implemented in C++. For the details on these patterns, you can check out the design patterns documentation.

Section 112.1: Adapter Pattern

Convert the interface of a class into another interface clients expect. Adapter (or Wrapper) lets classes work together that couldn't otherwise because of incompatible interfaces. Adapter pattern's motivation is that we can reuse existing software if we can modify the interface.

1. Adapter pattern relies on object composition.
2. Client calls operation on Adapter object.
3. Adapter calls Adaptee to carry out the operation.
4. In STL, stack adapted from vector: When stack executes push(), underlying vector does vector::push_back().

Example:

```
#include <iostream>

// Desired interface (Target)
class Rectangle
{
  public:
    virtual void draw() = 0;
};

// Legacy component (Adaptee)
class LegacyRectangle
{
  public:
    LegacyRectangle(int x1, int y1, int x2, int y2) {
        x1_ = x1;
        y1_ = y1;
        x2_ = x2;
        y2_ = y2;
        std::cout << "LegacyRectangle(x1,y1,x2,y2)\n";
    }
    void oldDraw() {
        std::cout << "LegacyRectangle:  oldDraw(). \n";
    }
  private:
    int x1_;
    int y1_;
    int x2_;
    int y2_;
};

// Adapter wrapper
class RectangleAdapter: public Rectangle, private LegacyRectangle
{
  public:
    RectangleAdapter(int x, int y, int w, int h):
      LegacyRectangle(x, y, x + w, y + h) {
```

```
        std::cout << "RectangleAdapter(x,y,x+w,x+h)\n";
      }

    void draw() {
        std::cout << "RectangleAdapter: draw().\n";
        oldDraw();
    }
};

int main()
{
  int x = 20, y = 50, w = 300, h = 200;
  Rectangle *r = new RectangleAdapter(x,y,w,h);
  r->draw();
}

//Output:
//LegacyRectangle(x1,y1,x2,y2)
//RectangleAdapter(x,y,x+w,x+h)
```

Summary of the code:

1. The client thinks he is talking to a `Rectangle`
2. The target is the `Rectangle` class. This is what the client invokes method on.

```
Rectangle *r = new RectangleAdapter(x,y,w,h);
r->draw();
```

3. Note that the adapter class uses multiple inheritance.

```
class RectangleAdapter: public Rectangle, private LegacyRectangle {
    ...
}
```

4. The Adapter `RectangleAdapter` lets the `LegacyRectangle` responds to request (`draw()` on a `Rectangle`) by inheriting BOTH classes.
5. The `LegacyRectangle` class does not have the same methods (`draw()`) as `Rectangle`, but the `Adapter(RectangleAdapter)` can take the `Rectangle` method calls and turn around and invoke method on the `LegacyRectangle`, `oldDraw()`.

```
class RectangleAdapter: public Rectangle, private LegacyRectangle {
  public:
    RectangleAdapter(int x, int y, int w, int h):
      LegacyRectangle(x, y, x + w, y + h) {
        std::cout << "RectangleAdapter(x,y,x+w,x+h)\n";
      }

    void draw() {
        std::cout << "RectangleAdapter: draw().\n";
        oldDraw();
    }
};
```

Adapter design pattern translates the interface for one class into a compatible but different interface. So, this is similar to the **proxy** pattern in that it's a single-component wrapper. But the interface for the adapter class and the

original class may be different.

As we've seen in the example above, this **adapter** pattern is useful to expose a different interface for an existing API to allow it to work with other code. Also, by using adapter pattern, we can take heterogeneous interfaces, and transform them to provide consistent API.

Bridge pattern has a structure similar to an object adapter, but Bridge has a different intent: It is meant to **separate** an interface from its implementation so that they can be varied easily and independently. An **adapter** is meant to **change the interface** of an **existing** object.

Section 112.2: Observer pattern

Observer Pattern's intent is to define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

The subject and observers define the one-to-many relationship. The observers are dependent on the subject such that when the subject's state changes, the observers get notified. Depending on the notification, the observers may also be updated with new values.

Here is the example from the book "Design Patterns" by Gamma.

```
#include <iostream>
#include <vector>

class Subject;

class Observer
{
public:
    virtual ~Observer() = default;
    virtual void Update(Subject&) = 0;
};

class Subject
{
public:
     virtual ~Subject() = default;
     void Attach(Observer& o) { observers.push_back(&o); }
     void Detach(Observer& o)
     {
         observers.erase(std::remove(observers.begin(), observers.end(), &o));
     }
     void Notify()
     {
         for (auto* o : observers) {
             o->Update(*this);
         }
     }
private:
     std::vector<Observer*> observers;
};

class ClockTimer : public Subject
{
public:

    void SetTime(int hour, int minute, int second)
    {
        this->hour = hour;
```

```
        this->minute = minute;
        this->second = second;

        Notify();
    }

    int GetHour() const { return hour; }
    int GetMinute() const { return minute; }
    int GetSecond() const { return second; }

private:
    int hour;
    int minute;
    int second;
};

class DigitalClock: public Observer
{
public:
     explicit DigitalClock(ClockTimer& s) : subject(s) { subject.Attach(*this); }
     ~DigitalClock() { subject.Detach(*this); }
     void Update(Subject& theChangedSubject) override
     {
         if (&theChangedSubject == &subject) {
             Draw();
         }
     }

     void Draw()
     {
         int hour = subject.GetHour();
         int minute = subject.GetMinute();
         int second = subject.GetSecond();

         std::cout << "Digital time is " << hour << ":"
                   << minute << ":"
                   << second << std::endl;
     }

private:
     ClockTimer& subject;
};

class AnalogClock: public Observer
{
public:
     explicit AnalogClock(ClockTimer& s) : subject(s) { subject.Attach(*this); }
     ~AnalogClock() { subject.Detach(*this); }
     void Update(Subject& theChangedSubject) override
     {
         if (&theChangedSubject == &subject) {
             Draw();
         }
     }
     void Draw()
     {
         int hour = subject.GetHour();
         int minute = subject.GetMinute();
         int second = subject.GetSecond();

         std::cout << "Analog time is " << hour << ":"
                   << minute << ":"
```

```
                    << second << std::endl;
    }
private:
     ClockTimer& subject;
};

int main()
{
    ClockTimer timer;

    DigitalClock digitalClock(timer);
    AnalogClock analogClock(timer);

    timer.SetTime(14, 41, 36);
}
```

Output:

```
Digital time is 14:41:36
Analog time is 14:41:36
```

Here are the summary of the pattern:

1. Objects (`DigitalClock` or `AnalogClock` object) use the Subject interfaces (`Attach()` or `Detach()`) either to subscribe (register) as observers or unsubscribe (remove) themselves from being observers (`subject.Attach(*this);` , `subject.Detach(*this);`.
2. Each subject can have many observers(`vector<Observer*> observers;`).
3. All observers need to implement the Observer interface. This interface just has one method, `Update()`, that gets called when the Subject's state changes (`Update(Subject &)`)
4. In addition to the `Attach()` and `Detach()` methods, the concrete subject implements a `Notify()` method that is used to update all the current observers whenever state changes. But in this case, all of them are done in the parent class, `Subject` (`Subject::Attach (Observer&)`, `void Subject::Detach(Observer&)` and `void Subject::Notify()` .
5. The Concrete object may also have methods for setting and getting its state.
6. Concrete observers can be any class that implements the Observer interface. Each observer subscribe (register) with a concrete subject to receive update (`subject.Attach(*this);`).
7. The two objects of Observer Pattern are **loosely coupled**, they can interact but with little knowledge of each other.

Variation:

Signal and Slots

Signals and slots is a language construct introduced in Qt, which makes it easy to implement the Observer pattern while avoiding boilerplate code. The concept is that controls (also known as widgets) can send signals containing event information which can be received by other controls using special functions known as slots. The slot in Qt must be a class member declared as such. The signal/slot system fits well with the way Graphical User Interfaces are designed. Similarly, the signal/slot system can be used for asynchronous I/O (including sockets, pipes, serial devices, etc.) event notification or to associate timeout events with appropriate object instances and methods or functions. No registration/deregistration/invocation code need be written, because Qt's Meta Object Compiler (MOC) automatically generates the needed infrastructure.

The C# language also supports a similar construct although with a different terminology and syntax: events play the role of signals, and delegates are the slots. Additionally, a delegate can be a local variable, much like a function pointer, while a slot in Qt must be a class member declared as such.

Section 112.3: Factory Pattern

Factory pattern decouples object creation and allows creation by name using a common interface:

```
class Animal{
public:
    virtual std::shared_ptr<Animal> clone() const = 0;
    virtual std::string  getname() const = 0;
};

class Bear: public Animal{
public:
    virtual std::shared_ptr<Animal> clone() const override
    {
        return std::make_shared<Bear>(*this);
    }
    virtual std::string getname() const override
    {
        return "bear";
    }
};


class Cat: public Animal{
public:
    virtual std::shared_ptr<Animal> clone() const override
    {
        return std::make_shared<Cat>(*this);
    }
    virtual std::string  getname() const override
    {
        return "cat";
    }
};

class AnimalFactory{
public:
    static std::shared_ptr<Animal> getAnimal( const std::string&   name )
    {
      if ( name == "bear" )
        return std::make_shared<Bear>();
      if ( name == "cat" )
        return std::shared_ptr<Cat>();

    return nullptr;
    }


};
```

Section 112.4: Builder Pattern with Fluent API

The Builder Pattern decouples the creation of the object from the object itself. The main idea behind is that **an object does not have to be responsible for its own creation**. The correct and valid assembly of a complex object may be a complicated task in itself, so this task can be delegated to another class.

Inspired by the Email Builder in C#, I've decided to make a C++ version here. An Email object is not necessarily a *very complex object*, but it can demonstrate the pattern.

```
#include <iostream>
#include <sstream>
#include <string>

using namespace std;

// Forward declaring the builder
class EmailBuilder;

class Email
{
  public:
    friend class EmailBuilder;  // the builder can access Email's privates

    static EmailBuilder make();

    string to_string() const {
        stringstream stream;
        stream << "from: " << m_from
               << "\nto: " << m_to
               << "\nsubject: " << m_subject
               << "\nbody: " << m_body;
        return stream.str();
    }

  private:
    Email() = default; // restrict construction to builder

    string m_from;
    string m_to;
    string m_subject;
    string m_body;
};

class EmailBuilder
{
  public:
    EmailBuilder& from(const string &from) {
        m_email.m_from = from;
        return *this;
    }

    EmailBuilder& to(const string &to) {
        m_email.m_to = to;
        return *this;
    }

    EmailBuilder& subject(const string &subject) {
        m_email.m_subject = subject;
        return *this;
    }

    EmailBuilder& body(const string &body) {
        m_email.m_body = body;
        return *this;
    }

    operator Email&&() {
```

```
        return std::move(m_email); // notice the move
    }

  private:
    Email m_email;
};

EmailBuilder Email::make()
{
    return EmailBuilder();
}

// Bonus example!
std::ostream& operator <<(std::ostream& stream, const Email& email)
{
    stream << email.to_string();
    return stream;
}


int main()
{
    Email mail = Email::make().from("me@mail.com")
                              .to("you@mail.com")
                              .subject("C++ builders")
                              .body("I like this API, don't you?");

    cout << mail << endl;
}
```

For older versions of C++, one may just ignore the `std::move` operation and remove the && from the conversion operator (although this will create a temporary copy).

The builder finishes its work when it releases the built email by the `operator Email&&()`. In this example, the builder is a temporary object and returns the email before being destroyed. You could also use an explicit operation like `Email EmailBuilder::build() {...}` instead of the conversion operator.

Pass the builder around

A great feature the Builder Pattern provides is the ability to **use several actors to build an object together.** This is done by passing the builder to the other actors that will each one give some more information to the built object. This is specially powerful when you are building some sort of query, adding filters and other specifications.

```
void add_addresses(EmailBuilder& builder)
{
    builder.from("me@mail.com")
           .to("you@mail.com");
}

void compose_mail(EmailBuilder& builder)
{
    builder.subject("I know the subject")
           .body("And the body. Someone else knows the addresses.");
}

int main()
{
    EmailBuilder builder;
    add_addresses(builder);
    compose_mail(builder);
```

```
    Email mail = builder;
    cout << mail << endl;
}
```

Design variant : Mutable object

You can change the design of this pattern to fit your needs. I'll give one variant.

In the given example the Email object is immutable, i.e., it's properties can't be modified because there is no access to them. This was a desired feature. If you need to modify the object after its creation you have to provide some setters to it. Since those setters would be duplicated in the builder, you may consider to do it all in one class (no builder class needed anymore). Nevertheless, I would consider the need to make the built object mutable in the first place.

Chapter 113: Singleton Design Pattern

Section 113.1: Lazy Initialization

This example has been lifted from the `Q & A` section here:http://stackoverflow.com/a/1008289/3807729

See this article for a simple design for a lazy evaluated with guaranteed destruction singleton:
Can any one provide me a sample of Singleton in c++?

The classic lazy evaluated and correctly destroyed singleton.

```
class S
{
    public:
        static S& getInstance()
        {
            static S    instance; // Guaranteed to be destroyed.
                                  // Instantiated on first use.
            return instance;
        }
    private:
        S() {};                   // Constructor? (the {} brackets) are needed here.

        // C++ 03
        // ========
        // Don't forget to declare these two. You want to make sure they
        // are unacceptable otherwise you may accidentally get copies of
        // your singleton appearing.
        S(S const&);              // Don't Implement
        void operator=(S const&); // Don't implement

        // C++ 11
        // =======
        // We can use the better technique of deleting the methods
        // we don't want.
    public:
        S(S const&)               = delete;
        void operator=(S const&)  = delete;

        // Note: Scott Meyers mentions in his Effective Modern
        //       C++ book, that deleted functions should generally
        //       be public as it results in better error messages
        //       due to the compilers behavior to check accessibility
        //       before deleted status
};
```

See this article about when to use a singleton: (not often)
Singleton: How should it be used

See this two article about initialization order and how to cope:
Static variables initialisation order
Finding C++ static initialization order problems

See this article describing lifetimes:
What is the lifetime of a static variable in a C++ function?

See this article that discusses some threading implications to singletons:
Singleton instance declared as static variable of GetInstance method

See this article that explains why double checked locking will not work on C++:
What are all the common undefined behaviours that a C++ programmer should know about?

Section 113.2: Static deinitialization-safe singleton

There are times with multiple static objects where you need to be able to guarantee that the *singleton* will not be destroyed until all the static objects that use the *singleton* no longer need it.

In this case `std::shared_ptr` can be used to keep the *singleton* alive for all users even when the static destructors are being called at the end of the program:

```
class Singleton
{
public:
    Singleton(Singleton const&) = delete;
    Singleton& operator=(Singleton const&) = delete;

    static std::shared_ptr<Singleton> instance()
    {
        static std::shared_ptr<Singleton> s{new Singleton};
        return s;
    }

private:
    Singleton() {}
};
```

NOTE: This example appears as an answer in the Q&A section here.

Section 113.3: Thread-safe Singeton

Version ≥ C++11

The C++11 standards guarantees that the initialization of function scope objects are initialized in a synchronized manner. This can be used to implement a thread-safe singleton with lazy initialization.

```
class Foo
{
public:
    static Foo& instance()
    {
        static Foo inst;
        return inst;
    }
private:
    Foo() {}
    Foo(const Foo&) = delete;
    Foo& operator =(const Foo&) = delete;
};
```

Section 113.4: Subclasses

```
class API
{
public:
    static API& instance();

    virtual ~API() {}
```

```
    virtual const char* func1() = 0;
    virtual void func2() = 0;

protected:
    API() {}
    API(const API&) = delete;
    API& operator=(const API&) = delete;
};

class WindowsAPI : public API
{
public:
    virtual const char* func1()  override { /* Windows code */ }
    virtual void func2() override { /* Windows code */ }
};

class LinuxAPI : public API
{
public:
    virtual const char* func1() override { /* Linux code */ }
    virtual void func2() override { /* Linux code */ }
};

API& API::instance() {
#if PLATFORM == WIN32
    static WindowsAPI instance;
#elif PLATFORM = LINUX
    static LinuxAPI instance;
#endif
    return instance;
}
```

In this example, a simple compiler switch binds the `API` class to the appropriate subclass. In this way, `API` can be accessed without being coupled to platform-specific code.

Chapter 114: User-Defined Literals

Section 114.1: Self-made user-defined literal for binary

Despite you can write a binary number in C++14 like:

```
int number =0b0001'0101; // ==21
```

here comes a famous example with a self-made implementation for binary numbers:

Note: The whole template expanding program is running at compile time.

```
template< char FIRST, char... REST > struct binary
{
    static_assert( FIRST == '0' || FIRST == '1', "invalid binary digit" ) ;
    enum { value = ( ( FIRST - '0' ) << sizeof...(REST) ) + binary<REST...>::value  } ;
};

template<> struct binary<'0'> { enum { value = 0 } ; };
template<> struct binary<'1'> { enum { value = 1 } ; };


// raw literal operator
template<  char... LITERAL > inline
constexpr unsigned int operator "" _b() { return binary<LITERAL...>::value ; }

// raw literal operator
template<  char... LITERAL > inline
constexpr unsigned int operator "" _B() { return binary<LITERAL...>::value ; }

#include <iostream>

int main()
{
    std::cout  << 10101_B << ", " << 011011000111_b  << '\n' ; // prints 21, 1735
}
```

Section 114.2: Standard user-defined literals for duration

Version ≥ C++14

Those following duration user literals are declared in the `namespace std::literals::chrono_literals`, where both `literals` and `chrono_literals` are inline namespaces. Access to these operators can be gained with `using namespace std::literals`, `using namespace std::chrono_literals`, and `using namespace std::literals::chrono_literals`.

```
#include <chrono>
#include <iostream>

int main()
{
    using namespace std::literals::chrono_literals;

    std::chrono::nanoseconds t1 = 600ns;
    std::chrono::microseconds t2 = 42us;
    std::chrono::milliseconds t3 = 51ms;
    std::chrono::seconds t4 = 61s;
    std::chrono::minutes t5 = 88min;
```

```
    auto t6 = 2 * 0.5h;

    auto total = t1 + t2 + t3 + t4 + t5 + t6;

    std::cout.precision(13);
    std::cout << total.count() << " nanoseconds" << std::endl; // 8941051042600 nanoseconds
    std::cout << std::chrono::duration_cast<std::chrono::hours>(total).count()
              << " hours" << std::endl; // 2 hours
}
```

Section 114.3: User-defined literals with long double values

```
#include <iostream>

long double operator"" _km(long double val)
{
    return val * 1000.0;
}

long double operator"" _mi(long double val)
{
    return val * 1609.344;
}

int main()
{
    std::cout << "3 km = " << 3.0_km << " m\n";
    std::cout << "3 mi = " << 3.0_mi << " m\n";
    return 0;
}
```

The output of this program is the following:

```
3 km = 3000 m
3 mi = 4828.03 m
```

Section 114.4: Standard user-defined literals for strings

Version ≥ C++14

Those following string user literals are declared in the `namespace std::literals::string_literals`, where both `literals` and `string_literals` are inline namespaces. Access to these operators can be gained with `using namespace std::literals`, `using namespace std::string_literals`, and `using namespace std::literals::string_literals`.

```
#include <codecvt>
#include <iostream>
#include <locale>
#include <string>

int main()
{
    using namespace std::literals::string_literals;

    std::string s = "hello world"s;
    std::u16string s16 = u"hello world"s;
    std::u32string s32 = U"hello world"s;
    std::wstring ws = L"hello world"s;
```

```
    std::cout << s << std::endl;

    std::wstring_convert<std::codecvt_utf8_utf16<char16_t>, char16_t> utf16conv;
    std::cout << utf16conv.to_bytes(s16) << std::endl;

    std::wstring_convert<std::codecvt_utf8_utf16<char32_t>, char32_t> utf32conv;
    std::cout << utf32conv.to_bytes(s32) << std::endl;

    std::wcout << ws << std::endl;
}
```

Note:

Literal string may containing `\0`

```
std::string s1 = "foo\0\0bar";  // constructor from C-string: results in "foo"s
std::string s2 = "foo\0\0bar"s; // That string contains 2 '\0' in its middle
```

Section 114.5: Standard user-defined literals for complex

Version ≥ C++14

Those following complex user literals are declared in the `namespace std::literals::complex_literals`, where both `literals` and `complex_literals` are inline namespaces. Access to these operators can be gained with `using namespace std::literals`, `using namespace std::complex_literals`, and `using namespace std::literals::complex_literals`.

```
#include <complex>
#include <iostream>

int main()
{
    using namespace std::literals::complex_literals;

    std::complex<double> c = 2.0 + 1i;        // {2.0, 1.}
    std::complex<float> cf = 2.0f + 1if;      // {2.0f, 1.f}
    std::complex<long double> cl = 2.0L + 1il; // {2.0L, 1.L}

    std::cout << "abs" << c << " = " << abs(c) << std::endl;   // abs(2,1) = 2.23607
    std::cout << "abs" << cf << " = " << abs(cf) << std::endl; // abs(2,1) = 2.23607
    std::cout << "abs" << cl << " = " << abs(cl) << std::endl; // abs(2,1) = 2.23607
}
```

Chapter 115: Memory management

Section 115.1: Free Storage (Heap, Dynamic Allocation ...)

The term **'heap'** is a general computing term meaning an area of memory from which portions can be allocated and deallocated independently of the memory provided by the **stack**.

In `C++` the *Standard* refers to this area as the **Free Store** which is considered a more accurate term.

Areas of memory allocated from the **Free Store** may live longer than the original scope in which it was allocated. Data too large to be stored on the stack may also be allocated from the **Free Store**.

Raw memory can be allocated and deallocated by the *new* and *delete* keywords.

```
float *foo = nullptr;
{
    *foo = new float; // Allocates memory for a float
    float bar;              // Stack allocated
} // End lifetime of bar, while foo still alive

delete foo;          // Deletes the memory for the float at pF, invalidating the pointer
foo = nullptr;       // Setting the pointer to nullptr after delete is often considered good
practice
```

It's also possible to allocate fixed size arrays with *new* and *delete*, with a slightly different syntax. Array allocation is not compatible with non-array allocation, and mixing the two will lead to heap corruption. Allocating an array also allocates memory to track the size of the array for later deletion in an implementation-defined way.

```
// Allocates memory for an array of 256 ints
int *foo = new int[256];
// Deletes an array of 256 ints at foo
delete[] foo;
```

When using *new* and *delete* instead *malloc* and *free*, the constructor and destructor will get executed (Similar to stack based objects). This is why *new* and *delete* are preferred over *malloc* and *free*.

```
struct ComplexType {
    int a = 0;

    ComplexType() { std::cout << "Ctor" << std::endl; }
    ~ComplexType() { std::cout << "Dtor" << std::endl; }
};

// Allocates memory for a ComplexType, and calls its constructor
ComplexType *foo = new ComplexType();
//Calls the destructor for ComplexType() and deletes memory for a Complextype at pC
delete foo;
```

Version ≥ C++11

From C++11 on, the use of smart pointers is recommended for indicating ownership.

Version ≥ C++14

C++14 added `std::make_unique` to the STL, changing the recommendation to favor `std::make_unique` or `std::make_shared` instead of using naked *new* and *delete*.

Section 115.2: Placement new

There are situations when we don't want to rely upon Free Store for allocating memory and we want to use custom memory allocations using `new`.

For these situations we can use `Placement New`, where we can tell `new' operator to allocate memory from a pre-allocated memory location

For example

```
int a4byteInteger;

char *a4byteChar = new (&a4byteInteger) char[4];
```

In this example, the memory pointed by `a4byteChar` is 4 byte allocated to 'stack' via integer variable `a4byteInteger`.

The benefit of this kind of memory allocation is the fact that programmers control the allocation. In the example above, since `a4byteInteger` is allocated on stack, we don't need to make an explicit call to 'delete a4byteChar`.

Same behavior can be achieved for dynamic allocated memory also. For example

```
int *a8byteDynamicInteger = new int[2];

char *a8byteChar = new (a8byteDynamicInteger) char[8];
```

In this case, the memory pointer by `a8byteChar` will be referring to dynamic memory allocated by `a8byteDynamicInteger`. In this case however, we need to explicitly call`delete a8byteDynamicInteger` to release the memory

Another example for C++ Class

```
struct ComplexType {
    int a;

    ComplexType() : a(0) {}
    ~ComplexType() {}
};

int main() {
    char* dynArray = new char[256];

    //Calls ComplexType's constructor to initialize memory as a ComplexType
    new((void*)dynArray) ComplexType();

    //Clean up memory once we're done
    reinterpret_cast<ComplexType*>(dynArray)->~ComplexType();
    delete[] dynArray;

    //Stack memory can also be used with placement new
    alignas(ComplexType) char localArray[256]; //alignas() available since C++11

    new((void*)localArray) ComplexType();

    //Only need to call the destructor for stack memory
    reinterpret_cast<ComplexType*>(localArray)->~ComplexType();

    return 0;
}
```

Section 115.3: Stack

The stack is a small region of memory into which temporary values are placed during execution. Allocating data into the stack is very fast compared to heap allocation, as all the memory has already been assigned for this purpose.

```
int main() {
    int a = 0; //Stored on the stack
    return a;
}
```

The stack is named because chains of function calls will have their temporary memory 'stacked' on top of each other, each one using a separate small section of memory.

```
float bar() {
    //f will be placed on the stack after anything else
    float f = 2;
    return f;
}

double foo() {
    //d will be placed just after anything within main()
    double d = bar();
    return d;
}

int main() {
    //The stack has no user variables stored in it until foo() is called
    return (int)foo();
}
```

Data stored on the stack is only valid so long as the scope that allocated the variable is still active.

```
int* pA = nullptr;

void foo() {
    int b = *pA;
    pA = &b;
}

int main() {
    int a = 5;
    pA = &a;
    foo();
    //Undefined behavior, the value pointed to by pA is no longer in scope
    a = *pA;
}
```

Chapter 116: C++11 Memory Model

Different threads trying to access the same memory location participate in a *data race* if at least one of the operations is a modification (also known as *store operation*). These *data races* cause *undefined behavior*. To avoid them one needs to prevent these threads from concurrently executing such conflicting operations.

Synchronization primitives (mutex, critical section and the like) can guard such accesses. The Memory Model introduced in C++11 defines two new portable ways to synchronize access to memory in multi-threaded environment: atomic operations and fences.

Atomic Operations

It is now possible to read and write to given memory location by the use of *atomic load* and *atomic store* operations. For convenience these are wrapped in the `std::atomic<t>` template class. This class wraps a value of type `t` but this time *loads* and *stores* to the object are atomic.

The template is not available for all types. Which types are available is implementation specific, but this usually includes most (or all) available integral types as well as pointer types. So that `std::atomic<unsigned>` and `std::atomic<std::vector<foo> *>` should be available, while `std::atomic<std::pair<bool,char>>` most probably wont be.

Atomic operations have the following properties:

- All atomic operations can be performed concurrently from multiple threads without causing undefined behavior.
- An *atomic load* will see either the initial value which the atomic object was constructed with, or the value written to it via some *atomic store* operation.
- *Atomic stores* to the same atomic object are ordered the same in all threads. If a thread has already seen the value of some *atomic store* operation, subsequent *atomic load* operations will see either the same value, or the value stored by subsequent *atomic store* operation.
- *Atomic read-modify-write* operations allow *atomic load* and *atomic store* to happen without other *atomic store* in between. For example one can atomically increment a counter from multiple threads, and no increment will be lost regardless of the contention between the threads.
- Atomic operations receive an optional `std::memory_order` parameter which defines what additional properties the operation has regarding other memory locations.

std::memory_order	Meaning
`std::memory_order_relaxed`	no additional restrictions
`std::memory_order_release` → `std::memory_order_acquire`	if `load-acquire` sees the value stored by `store-release` then stores *sequenced before* the `store-release` happen before loads sequenced after the *load acquire*
`std::memory_order_consume`	like `memory_order_acquire` but only for dependent loads
`std::memory_order_acq_rel`	combines `load-acquire` and `store-release`
`std::memory_order_seq_cst`	sequential consistency

These memory order tags allow three different memory ordering disciplines: *sequential consistency*, *relaxed*, and *release-acquire* with its sibling *release-consume*.

Sequential Consistency

If no memory order is specified for an atomic operation, the order defaults to *sequential consistency*. This mode can also be explicitly selected by tagging the operation with `std::memory_order_seq_cst`.

With this order no memory operation can cross the atomic operation. All memory operations sequenced before the atomic operation happen before the atomic operation and the atomic operation happens before all memory operations that are sequenced after it. This mode is probably the easiest one to reason about but it also leads to the greatest penalty to performance. It also prevents all compiler optimizations that might otherwise try to reorder operations past the atomic operation.

Relaxed Ordering

The opposite to *sequential consistency* is the *relaxed* memory ordering. It is selected with the `std::memory_order_relaxed` tag. Relaxed atomic operation will impose no restrictions on other memory operations. The only effect that remains, is that the operation is itself still atomic.

Release-Acquire Ordering

An *atomic store* operation can be tagged with `std::memory_order_release` and an *atomic load* operation can be tagged with `std::memory_order_acquire`. The first operation is called *(atomic) store-release* while the second is called *(atomic) load-acquire*.

When *load-acquire* sees the value written by a *store-release* the following happens: all store operations sequenced before the *store-release* become visible to (*happen before*) load operations that are sequenced after the *load-acquire*.

Atomic read-modify-write operations can also receive the cumulative tag `std::memory_order_acq_rel`. This makes the *atomic load* portion of the operation an *atomic load-acquire* while the *atomic store* portion becomes *atomic store-release*.

The compiler is not allowed to move store operations after an *atomic store-release* operation. It is also not allowed to move load operations before *atomic load-acquire* (or *load-consume*).

Also note that there is no *atomic load-release* or *atomic store-acquire*. Attempting to create such operations makes them *relaxed* operations.

Release-Consume Ordering

This combination is similar to *release-acquire*, but this time the *atomic load* is tagged with `std::memory_order_consume` and becomes *(atomic) load-consume* operation. This mode is the same as *release-acquire* with the only difference that among the load operations sequenced after the *load-consume* only these depending on the value loaded by the *load-consume* are ordered.

Fences

Fences also allow memory operations to be ordered between threads. A fence is either a release fence or acquire fence.

If a release fence happens before an acquire fence, then stores sequenced before the release fence are visible to loads sequenced after the acquire fence. To guarantee that the release fence happens before the acquire fence one may use other synchronization primitives including relaxed atomic operations.

Section 116.1: Need for Memory Model

```
int x, y;
bool ready = false;

void init()
{
  x = 2;
```

```
  y = 3;
  ready = true;
}
void use()
{
  if (ready)
    std::cout << x + y;
}
```

One thread calls the `init()` function while another thread (or signal handler) calls the `use()` function. One might expect that the `use()` function will either print `5` or do nothing. This may not always be the case for several reasons:

- The CPU may reorder the writes that happen in `init()` so that the code that actually executes might look like:

```
void init()
{
  ready = true;
  x = 2;
  y = 3;
}
```

- The CPU may reorder the reads that happen in `use()` so that the actually executed code might become:

```
void use()
{
  int local_x = x;
  int local_y = y;
  if (ready)
    std::cout << local_x + local_y;
}
```

- An optimizing C++ compiler may decide to reorder the program in similar way.

Such reordering cannot change the behavior of a program running in single thread because a thread cannot interleave the calls to `init()` and `use()`. On the other hand in a multi-threaded setting one thread may see part of the writes performed by the other thread where it may happen that `use()` may see `ready==true` and garbage in `x` or `y` or both.

The C++ Memory Model allows the programmer to specify which reordering operations are permitted and which are not, so that a multi-threaded program would also be able to behave as expected. The example above can be rewritten in thread-safe way like this:

```
int x, y;
std::atomic<bool> ready{false};

void init()
{
  x = 2;
  y = 3;
  ready.store(true, std::memory_order_release);
}
void use()
{
  if (ready.load(std::memory_order_acquire))
    std::cout << x + y;
}
```

Here `init()` performs *atomic store-release* operation. This not only stores the value `true` into `ready`, but also tells the compiler that it cannot move this operation before write operations that are *sequenced before* it.

The `use()` function does an *atomic load-acquire* operation. It reads the current value of `ready` and also forbids the compiler from placing read operations that are *sequenced after* it to *happen before* the *atomic load-acquire*.

These atomic operations also cause the compiler to put whatever hardware instructions are needed to inform the CPU to refrain from the unwanted reorderings.

Because the *atomic store-release* is to the same memory location as the *atomic load-acquire*, the memory model stipulates that if the *load-acquire* operation sees the value written by the *store-release* operation, then all writes performed by `init()`'s thread prior to that *store-release* will be visible to loads that `use()`'s thread executes after its *load-acquire*. That is if `use()` sees `ready==true`, then it is guaranteed to see `x==2` and `y==3`.

Note that the compiler and the CPU are still allowed to write to `y` before writing to `x`, and similarly the reads from these variables in `use()` can happen in any order.

Section 116.2: Fence example

The example above can also be implemented with fences and relaxed atomic operations:

```
int x, y;
std::atomic<bool> ready{false};

void init()
{
  x = 2;
  y = 3;
  atomic_thread_fence(std::memory_order_release);
  ready.store(true, std::memory_order_relaxed);
}
void use()
{
  if (ready.load(std::memory_order_relaxed))
  {
    atomic_thread_fence(std::memory_order_acquire);
    std::cout << x + y;
  }
}
```

If the atomic load operation sees the value written by the atomic store then the store happens before the load, and so do the fences: the release fence happens before the acquire fence making the writes to `x` and `y` that precede the release fence to become visible to the `std::cout` statement that follows the acquire fence.

A fence might be beneficial if it can reduce the overall number of acquire, release or other synchronization operations. For example:

```
void block_and_use()
{
  while (!ready.load(std::memory_order_relaxed))
    ;
  atomic_thread_fence(std::memory_order_acquire);
  std::cout << x + y;
}
```

The `block_and_use()` function spins until the `ready` flag is set with the help of relaxed atomic load. Then a single acquire fence is used to provide the needed memory ordering.

Chapter 117: Scopes

Section 117.1: Global variables

To declare a single instance of a variable which is accessible in different source files, it is possible to make it in the global scope with keyword `extern`. This keyword says the compiler that somewhere in the code there is a definition for this variable, so it can be used everywhere and all write/read will be done in one place of memory.

```
// File my_globals.h:

#ifndef __MY_GLOBALS_H__
#define __MY_GLOBALS_H__

extern int circle_radius; // Promise to the compiler that circle_radius
                          // will be defined somewhere

#endif
```

```
// File foo1.cpp:

#include "my_globals.h"

int circle_radius = 123; // Defining the extern variable
```

```
// File main.cpp:

#include "my_globals.h"
#include <iostream>

int main()
{
    std::cout << "The radius is: " << circle_radius << "\n";'
    return 0;
}
```

Output:

```
The radius is: 123
```

Section 117.2: Simple block scope

The scope of a variable in a block `{ ... }`, begins after declaration and ends at the end of the block. If there is nested block, the inner block can hide the scope of a variable which is declared in the outer block.

```
{
    int x = 100;
    //   ^
    //   Scope of `x` begins here
    //
}   // <- Scope of `x` ends here
```

If a nested block starts within an outer block, a new declared variable with the same name which is before in the

outer class, hides the first one.

```
{
    int x = 100;

    {
        int x = 200;

        std::cout << x;  // <- Output is 200
    }

    std::cout << x;  // <- Output is 100
}
```

Chapter 118: static_assert

Parameter	Details
bool_constexpr	Expression to check
message	Message to print when *bool_constexpr* is *false*

Section 118.1: static_assert

Assertations mean that a condition should be checked and if it's false, it's an error. For `static_assert()`, this is done compile-time.

```
template<typename T>
T mul10(const T t)
{
    static_assert( std::is_integral<T>::value, "mul10() only works for integral types" );
    return (t << 3) + (t << 1);
}
```

A `static_assert()` has a mandatory first parameter, the condition, that is a bool constexpr. It *might* have a second parameter, the message, that is a string literal. From C++17, the second parameter is optional; before that, it's mandatory.

Version ≥ C++17

```
template<typename T>
T mul10(const T t)
{
    static_assert(std::is_integral<T>::value);
    return (t << 3) + (t << 1);
}
```

It is used when:

- In general, a verification at compile-time is required on some type on constexpr value
- A template function needs to verify certain properties of a type passed to it
- One wants to write test cases for:
    - template metafunctions
    - constexpr functions
    - macro metaprogramming
- Certain defines are required (for ex., C++ version)
- Porting legacy code, assertations on `sizeof(T)` (e.g., 32-bit int)
- Certain compiler features are required for the program to work (packing, empty base class optimization, etc.)

Note that `static_assert()` does not participate in SFINAE: thus, when additional overloads / specializations are possible, one should not use it instead of template metaprogramming techniques (like `std::enable_if<>`). It might be used in template code when the expected overload / specialization is already found, but further verifications are required. In such cases, it might provide more concrete error message(s) than relying on SFINAE for this.

Chapter 119: constexpr

`constexpr` is a keyword that can be used to mark a variable's value as a constant expression, a function as potentially usable in constant expressions, or (since C++17) an if statement as having only one of its branches selected to be compiled.

Section 119.1: constexpr variables

A variable declared `constexpr` is implicitly `const` and its value may be used as a constant expression.

Comparison with `#define`

A `constexpr` is type-safe replacement for `#define` based compile-time expressions. With `constexpr` the compile-time evaluated expression is replaced with the result. For example:

Version ≥ C++11

```
int main()
{
   constexpr int N = 10 + 2;
   cout << N;
}
```

will produce the following code:

```
cout << 12;
```

A pre-processor based compile-time macro would be different. Consider:

```
#define N 10 + 2

int main()
{
    cout << N;
}
```

will produce:

```
cout << 10 + 2;
```

which will obviously be converted to `cout << 10 + 2;`. However, the compiler would have to do more work. Also, it creates a problem if not used correctly.

For example (with `#define`):

```
cout << N * 2;
```

forms:

```
cout << 10 + 2 * 2; // 14
```

But a pre-evaluated `constexpr` would correctly give `24`.

Comparison with `const`

A `const` variable is a **variable** which needs memory for its storage. A `constexpr` does not. A `constexpr` produces compile time constant, which cannot be changed. You may argue that `const` may also not be changed. But consider:

```
int main()
{
   const int size1 = 10;
   const int size2 = abs(10);

   int arr_one[size1];
   int arr_two[size2];
}
```

With most compilers the second statement will fail (may work with GCC, for example). The size of any array, as you might know, has to be a constant expression (i.e. results in compile-time value). The second variable `size2` is assigned some value that is decided at runtime (even though you know it is `10`, for the compiler it is not compile-time).

This means that a `const` may or may not be a true compile-time constant. You cannot guarantee or enforce that a particular `const` value is absolutely compile-time. You may use `#define` but it has its own pitfalls.

Therefore simply use:

Version ≥ C++11

```
int main()
{
    constexpr int size = 10;

    int arr[size];
}
```

A `constexpr` expression must evaluate to a compile-time value. Thus, you cannot use:

Version ≥ C++11

```
constexpr int size = abs(10);
```

Unless the function (`abs`) is itself returning a `constexpr`.

All basic types can be initialized with `constexpr`.

Version ≥ C++11

```
constexpr bool FailFatal = true;
constexpr float PI = 3.14f;
constexpr char* site= "StackOverflow";
```

Interestingly, and conveniently, you may also use `auto`:

Version ≥ C++11

```
constexpr auto domain = ".COM";  // const char * const domain = ".COM"
constexpr auto PI = 3.14;        // constexpr double
```

Section 119.2: Static if statement

Version ≥ C++17

The `if constexpr` statement can be used to conditionally compile code. The condition must be a constant expression. The branch not selected is *discarded.* A discarded statement inside a template is not instantiated. For

example:

```
template<class T, class ... Rest>
void g(T &&p, Rest &&...rs)
{
  // ... handle p
  if constexpr (sizeof...(rs) > 0)
    g(rs...);  // never instantiated with an empty argument list
}
```

In addition, variables and functions that are odr-used only inside discarded statements are not required to be defined, and discarded `return` statements are not used for function return type deduction.

`if constexpr` is distinct from `#ifdef`. `#ifdef` conditionally compiles code, but only based on conditions that can be evaluated at preprocessing time. For example, `#ifdef` could not be used to conditionally compile code depending on the value of a template parameter. On the other hand, `if constexpr` cannot be used to discard syntactically invalid code, while `#ifdef` can.

```
if constexpr(false) {
    foobar;  // error; foobar has not been declared
    std::vector<int> v("hello, world");  // error; no matching constructor
}
```

Section 119.3: constexpr functions

A function that is declared `constexpr` is implicitly inline and calls to such a function potentially yield constant expressions. For example, the following function, if called with constant expression arguments, yields a constant expression too:

Version ≥ C++11

```
constexpr int Sum(int a, int b)
{
    return a + b;
}
```

Thus, the result of the function call may be used as an array bound or a template argument, or to initialize a `constexpr` variable:

Version ≥ C++11

```
int main()
{
    constexpr int S = Sum(10,20);

    int Array[S];
    int Array2[Sum(20,30)]; // 50 array size, compile time
}
```

Note that if you remove `constexpr` from function's return type specification, assignment to `S` will not work, as `S` is a `constexpr` variable, and must be assigned a compile-time const. Similarly, size of array will also not be a constant-expression, if function `Sum` is not `constexpr`.

Interesting thing about `constexpr` functions is that you may also use it like ordinary functions:

Version ≥ C++11

```
int a = 20;
auto sum = Sum(a, abs(-20));
```

`Sum` will not be a `constexpr` function now, it will be compiled as an ordinary function, taking variable (non-constant) arguments, and returning non-constant value. You need not to write two functions.

It also means that if you try to assign such call to a non-const variable, it won't compile:

Version ≥ C++11

```
int a = 20;
constexpr auto sum = Sum(a, abs(-20));
```

The reason is simple: `constexpr` must only be assigned a compile-time constant. However, the above function call makes `Sum` a non-`constexpr` (R-value is non-const, but L-value is declaring itself to be `constexpr`).

The `constexpr` function **must** also return a compile-time constant. Following will not compile:

Version ≥ C++11

```
constexpr int Sum(int a, int b)
{
    int a1 = a;     // ERROR
    return a + b;
}
```

Because `a1` is a non-constexpr *variable*, and prohibits the function from being a true `constexpr` function. Making it `constexpr` and assigning it `a` will also not work - since value of `a` (incoming parameter) is still not yet known:

Version ≥ C++11

```
constexpr int Sum(int a, int b)
{
   constexpr int a1 = a;     // ERROR
   ..
```

Furthermore, following will also not compile:

Version ≥ C++11

```
constexpr int Sum(int a, int b)
{
   return abs(a) + b; // or abs(a) + abs(b)
}
```

Since `abs(a)` is not a constant expression (even `abs(10)` will not work, since `abs` is not returning a `constexpr int` !

What about this?

Version ≥ C++11

```
constexpr int Abs(int v)
{
    return v >= 0 ? v : -v;
}

constexpr int Sum(int a, int b)
{
    return Abs(a) + b;
}
```

We crafted our own `Abs` function which is a `constexpr`, and the body of `Abs` also doesn't break any rule. Also, at the call site (inside `Sum`), the expression evaluates to a `constexpr`. Hence, the call to `Sum(-10, 20)` will be a compile-time constant expression resulting to `30`.

Chapter 120: One Definition Rule (ODR)

Section 120.1: ODR violation via overload resolution

Even with identical tokens for inline functions, ODR can be violated if lookup of names doesn't refer to the same entity. let's consider `func` in following:

- header.h

```
void overloaded(int);
inline void func() { overloaded('*'); }
```

- foo.cpp

```
#include "header.h"

void foo()
{
    func(); // `overloaded` refers to `void overloaded(int)`
}
```

- bar.cpp

```
void overloaded(char); // can come from other include
#include "header.h"

void bar()
{
    func(); // `overloaded` refers to `void overloaded(char)`
}
```

We have an ODR violation as `overloaded` refers to different entities depending of the translation unit.

Section 120.2: Multiply defined function

The most important consequence of the One Definition Rule is that non-inline functions with external linkage should only be defined once in a program, although they can be declared multiple times. Therefore, such functions should not be defined in headers, since a header can be included multiple times from different translation units.

`foo.h`:

```
#ifndef FOO_H
#define FOO_H
#include <iostream>
void foo() { std::cout << "foo"; }
void bar();
#endif
```

`foo.cpp`:

```
#include "foo.h"
void bar() { std:: cout << "bar"; }
```

`main.cpp`:

```
#include "foo.h"
int main() {
    foo();
    bar();
}
```

In this program, the function `foo` is defined in the header `foo.h`, which is included twice: once from `foo.cpp` and once from `main.cpp`. Each translation unit therefore contains its own definition of `foo`. Note that the include guards in `foo.h` do not prevent this from happening, since `foo.cpp` and `main.cpp` both *separately* include `foo.h`. The most likely result of trying to build this program is a link-time error identifying `foo` as having been multiply defined.

To avoid such errors, one should *declare* functions in headers and *define* them in the corresponding `.cpp` files, with some exceptions (see other examples).

Section 120.3: Inline functions

A function declared `inline` may be defined in multiple translation units, provided that all definitions are identical. It also must be defined in every translation unit in which it is used. Therefore, inline functions *should* be defined in headers and there is no need to mention them in the implementation file.

The program will behave as though there is a single definition of the function.

`foo.h`:

```
#ifndef FOO_H
#define FOO_H
#include <iostream>
inline void foo() { std::cout << "foo"; }
void bar();
#endif
```

`foo.cpp`:

```
#include "foo.h"
void bar() {
    // more complicated definition
}
```

`main.cpp`:

```
#include "foo.h"
int main() {
    foo();
    bar();
}
```

In this example, the simpler function `foo` is defined inline in the header file while the more complicated function `bar` is not inline and is defined in the implementation file. Both the `foo.cpp` and `main.cpp` translation units contain definitions of `foo`, but this program is well-formed since `foo` is inline.

A function defined within a class definition (which may be a member function or a friend function) is *implicitly* inline. Therefore, if a class is defined in a header, member functions of the class may be defined within the class definition, even though the definitions may be included in multiple translation units:

```
// in foo.h
class Foo {
```

```
    void bar() { std::cout << "bar"; }
    void baz();
};

// in foo.cpp
void Foo::baz() {
   // definition
}
```

The function `Foo::baz` is defined out-of-line, so it is *not* an inline function, and must not be defined in the header.

Chapter 121: Unspecified behavior

Section 121.1: Value of an out-of-range enum

If a scoped enum is converted to an integral type that is too small to hold its value, the resulting value is unspecified. Example:

```
enum class E {
    X = 1,
    Y = 1000,
};
// assume 1000 does not fit into a char
char c1 = static_cast<char>(E::X); // c1 is 1
char c2 = static_cast<char>(E::Y); // c2 has an unspecified value
```

Also, if an integer is converted to an enum and the integer's value is outside the range of the enum's values, the resulting value is unspecified. Example:

```
enum Color {
    RED = 1,
    GREEN = 2,
    BLUE = 3,
};
Color c = static_cast<Color>(4);
```

However, in the next example, the behavior is *not* unspecified, since the source value is within the *range* of the enum, although it is unequal to all enumerators:

```
enum Scale {
    ONE = 1,
    TWO = 2,
    FOUR = 4,
};
Scale s = static_cast<Scale>(3);
```

Here `s` will have the value 3, and be unequal to `ONE`, `TWO`, and `FOUR`.

Section 121.2: Evaluation order of function arguments

If a function has multiple arguments, it is unspecified what order they are evaluated in. The following code could print `x = 1, y = 2` or `x = 2, y = 1` but it is unspecified which.

```
int f(int x, int y) {
    printf("x = %d, y = %d\n", x, y);
}
int get_val() {
    static int x = 0;
    return ++x;
}
int main() {
    f(get_val(), get_val());
}
```

Version ≥ C++17

In C++17, the order of evaluation of function arguments remains unspecified.

However, each function argument is completely evaluated, and the calling object is guaranteed evaluated before any function arguments are.

```
struct from_int {
  from_int(int x) { std::cout << "from_int (" << x << ")\n"; }
};
int make_int(int x){ std::cout << "make_int (" << x << ")\n"; return x; }


void foo(from_int a, from_int b) {
}
void bar(from_int a, from_int b) {
}

auto which_func(bool b){
  std::cout << b?"foo":"bar" << "\n";
  return b?foo:bar;
}

int main(int argc, char const*const* argv) {
  which_func( true )( make_int(1), make_int(2) );
}
```

this must print:

```
bar
make_int(1)
from_int(1)
make_int(2)
from_int(2)
```

or

```
bar
make_int(2)
from_int(2)
make_int(1)
from_int(1)
```

it may *not* print `bar` after any of the `make` or `from`'s, and it may not print:

```
bar
make_int(2)
make_int(1)
from_int(2)
from_int(1)
```

or similar. Prior to C++17 printing `bar` after `make_ints` was legal, as was doing both `make_ints` prior to doing any `from_ints`.

Section 121.3: Result of some reinterpret_cast conversions

The result of a `reinterpret_cast` from one function pointer type to another, or one function reference type to another, is unspecified. Example:

```
int f();
```

```
auto fp = reinterpret_cast<int(*)(int)>(&f); // fp has unspecified value
```

Version ≤ C++03

The result of a `reinterpret_cast` from one object pointer type to another, or one object reference type to another, is unspecified. Example:

```
int x = 42;
char* p = reinterpret_cast<char*>(&x); // p has unspecified value
```

However, with most compilers, this was equivalent to `static_cast<char*>(static_cast<void*>(&x))` so the resulting pointer `p` pointed to the first byte of `x`. This was made the standard behavior in C++11. See type punning conversion for more details.

Section 121.4: Space occupied by a reference

A reference is not an object, and unlike an object, it is not guaranteed to occupy some contiguous bytes of memory. The standard leaves it unspecified whether a reference requires any storage at all. A number of features of the language conspire to make it impossible to portably examine any storage the reference might occupy:

- If `sizeof` is applied to a reference, it returns the size of the referenced type, thereby giving no information about whether the reference occupies any storage.
- Arrays of references are illegal, so it is not possible to examine the addresses of two consecutive elements of a hypothetical reference of arrays in order to determine the size of a reference.
- If the address of a reference is taken, the result is the address of the referent, so we cannot get a pointer to the reference itself.
- If a class has a reference member, attempting to extract the address of that member using `offsetof` yields undefined behavior since such a class is not a standard-layout class.
- If a class has a reference member, the class is no longer standard layout, so attempts to access any data used to store the reference results in undefined or unspecified behavior.

In practice, in some cases a reference variable may be implemented similarly to a pointer variable and hence occupy the same amount of storage as a pointer, while in other cases a reference may occupy no space at all since it can be optimized out. For example, in:

```
void f() {
    int x;
    int& r = x;
    // do something with r
}
```

the compiler is free to simply treat `r` as an alias for `x` and replace all occurrences of `r` in the rest of the function `f` with `x`, and not allocate any storage to hold `r`.

Section 121.5: Moved-from state of most standard library classes

Version ≥ C++11

All standard library containers are left in a *valid but unspecified* state after being moved from. For example, in the following code, v2 will contain `{1, 2, 3, 4}` after the move, but `v1` is not guaranteed to be empty.

```
int main() {
    std::vector<int> v1{1, 2, 3, 4};
    std::vector<int> v2 = std::move(v1);
```

```
}
```

Some classes do have a precisely defined moved-from state. The most important case is that of `std::unique_ptr<T>`, which is guaranteed to be null after being moved from.

Section 121.6: Result of some pointer comparisons

If two pointers are compared using `<`, `>`, `<=`, or `>=`, the result is unspecified in the following cases:

- The pointers point into different arrays. (A non-array object is considered an array of size 1.)

```
int x;
int y;
const bool b1 = &x < &y;            // unspecified
int a[10];
const bool b2 = &a[0] < &a[1];      // true
const bool b3 = &a[0] < &x;         // unspecified
const bool b4 = (a + 9) < (a + 10); // true
                                    // note: a+10 points past the end of the array
```

- The pointers point into the same object, but to members with different access control.

```
class A {
  public:
    int x;
    int y;
    bool f1() { return &x < &y; } // true; x comes before y
    bool f2() { return &x < &z; } // unspecified
  private:
    int z;
};
```

Section 121.7: Static cast from bogus void* value

If a `void*` value is converted to a pointer to object type, `T*`, but is not properly aligned for `T`, the resulting pointer value is unspecified. Example:

```
// Suppose that alignof(int) is 4
int x = 42;
void* p1 = &x;
// Do some pointer arithmetic...
void* p2 = static_cast<char*>(p1) + 2;
int* p3 = static_cast<int*>(p2);
```

The value of `p3` is unspecified because `p2` cannot point to an object of type `int`; its value is not a properly aligned address.

Section 121.8: Order of initialization of globals across TU

Whereas inside a Translation Unit, order of initialization of global variables is specified, order of initialization across Translation Units is unspecified.

So program with following files

- foo.cpp

```
#include <iostream>

int dummyFoo = ((std::cout << "foo"), 0);
```

- bar.cpp

```
#include <iostream>

int dummyBar = ((std::cout << "bar"), 0);
```

- main.cpp

```
int main() {}
```

might produce as output:

```
foobar
```

or

```
barfoo
```

That may lead to *Static Initialization Order Fiasco*.

Chapter 122: Argument Dependent Name Lookup

Section 122.1: What functions are found

Functions are found by first collecting a set of *"associated classes"* and *"associated namespaces"* that include one ore more of the following, depending on the argument type `T`. First, let us show the rules for classes, enumeration and class template specialization names.

- If `T` is a nested class, member enumeration, then the surrounding class of it.
- If `T` is an enumeration (it may *also* be a class member!), the innermost namespace of it.
- If `T` is a class (it may *also* be nested!), all its base classes and the class itself. The innermost namespace of all associated classes.
- If `T` is a `ClassTemplate<TemplateArguments>` (this is *also* a class!), the classes and namespaces associated with the template type arguments, the namespace of any template template argument and the surrounding class of any template template argument, if a template argument is a member template.

Now there are a few rules for builtin types as well

- If `T` is a pointer to `U` or array of `U`, the classes and namespaces associated with `U`. Example: `void (*fptr)(A); f(fptr);`, includes the namespaces and classes associated with `void(A)` (see next rule).
- If `T` is a function type, the classes and namespaces associated with parameter and return types. Example: `void(A)` would includes the namespaces and classes associated with `A`.
- If `T` is a pointer to member, the classes and namespaces associated with the member type (may apply to both pointer to member functions and pointer to data member!). Example: `B A::*p; void (A::*pf)(B); f(p); f(pf);` includes the namespaces and classes associated with `A`, `B`, `void(B)` (which applies bullet above for function types).

All functions and templates within all associated namespaces are found by argument dependent lookup. In addition, *namespace-scope friend functions declared in associated classes are found*, which are normally not visible. Using directives are ignored, however.

All of the following example calls are valid, without qualifying `f` by the namespace name in the call.

```
namespace A {
   struct Z { };
   namespace I { void g(Z); }
   using namespace I;

   struct X { struct Y { }; friend void f(Y) { } };
   void f(X p) { }
   void f(std::shared_ptr<X> p) { }
}

// example calls
f(A::X());
f(A::X::Y());
f(std::make_shared<A::X>());

g(A::Z()); // invalid: "using namespace I;" is ignored!
```

Chapter 123: Attributes

Section 123.1: [[fallthrough]]

Version ≥ C++17

Whenever a case is ended in a switch, the code of the next case will get executed. This last one can be prevented by using the ´break` statement. As this so-called fallthrough behavior can introduce bugs when not intended, several compilers and static analyzers give a warning on this.

From C++17 on, a standard attribute was introduced to indicate that the warning is not needed when the code is meant to fall through. Compilers can safely give warnings when a case is ended without break or [[fallthrough]] and has at least one statement.

```
switch(input) {
    case 2011:
    case 2014:
    case 2017:
        std::cout << "Using modern C++" << std::endl;
        [[fallthrough]]; // > No warning
    case 1998:
    case 2003:
        standard = input;
}
```

See the proposal for more detailed examples on how [[fallthrough]] can be used.

Section 123.2: [[nodiscard]]

Version ≥ C++17

The [[nodiscard]] attribute can be used to indicate that the return value of a function shouldn't be ignored when you do a function call. If the return value is ignored, the compiler should give a warning on this. The attribute can be added to:

- A function definition
- A type

Adding the attribute to a type has the same behaviour as adding the attribute to every single function which returns this type.

```
template<typename Function>
[[nodiscard]] Finally<std::decay_t<Function>> onExit(Function &&f);

void f(int &i) {
    assert(i == 0);                    // Just to make comments clear!
    ++i;                               // i == 1
    auto exit1 = onExit([&i]{ --i; }); // Reduce by 1 on exiting f()
    ++i;                               // i == 2
    onExit([&i]{ --i; });              // BUG: Reducing by 1 directly
                                       //      Compiler warning expected
    std::cout << i << std::end;        // Expected: 2, Real: 1
}
```

See the proposal for more detailed examples on how [[nodiscard]] can be used.

Note: The implementation details of `Finally/onExit` are omitted in the example, see Finally/ScopeExit.

Section 123.3: [[deprecated]] and [[deprecated("reason")]]

Version ≥ C++14

C++14 introduced a standard way of deprecating functions via attributes. `[[deprecated]]` can be used to indicate that a function is deprecated. `[[deprecated("reason")]]` allows adding a specific reason which can be shown by the compiler.

```
void function(std::unique_ptr<A> &&a);

// Provides specific message which helps other programmers fixing there code
[[deprecated("Use the variant with unique_ptr instead, this function will be removed in the next 
release")]]
void function(std::auto_ptr<A> a);

// No message, will result in generic warning if called.
[[deprecated]]
void function(A *a);
```

This attribute may be applied to:

- the declaration of a class
- a typedef-name
- a variable
- a non-static data member
- a function
- an enumeration
- a template specialization

(ref. c++14 standard draft: 7.6.5 Deprecated attribute)

Section 123.4: [[maybe_unused]]

The `[[maybe_unused]]` attribute is created for indicating in code that certain logic might not be used. This if often linked to preprocessor conditions where this might be used or might not be used. As compilers can give warnings on unused variables, this is a way of suppressing them by indicating intent.

A typical example of variables which are needed in debug builds while unneeded in production are return values indicating success. In the debug builds, the condition should be asserted, though in production these asserts have been removed.

```
[[maybe_unused]] auto mapInsertResult = configuration.emplace("LicenseInfo", 
stringifiedLicenseInfo);
assert(mapInsertResult.second); // We only get called during startup, so we can't be in the map
```

A more complex example are different kind of helper functions which are in an unnamed namespace. If these functions aren't used during compilation, a compiler might give a warning on them. Ideally you would like to guard them with the same preprocessor tags as the caller, though as this might become complex the `[[maybe_unused]]` attribute is a more maintainable alternative.

```
namespace {
    [[maybe_unused]] std::string createWindowsConfigFilePath(const std::string &relativePath);
```

```
    // TODO: Reuse this on BSD, MAC ...
    [[maybe_unused]] std::string createLinuxConfigFilePath(const std::string &relativePath);
}

std::string createConfigFilePath(const std::string &relativePath) {
#if OS == "WINDOWS"
      return createWindowsConfigFilePath(relativePath);
#elif OS == "LINUX"
      return createLinuxConfigFilePath(relativePath);
#else
#error "OS is not yet supported"
#endif
}
```

See the proposal for more detailed examples on how `[[maybe_unused]]` can be used.

Section 123.5: [[noreturn]]

Version ≥ C++11

C++11 introduced the `[[noreturn]]` attribute. It can be used for a function to indicate that the function does not return to the caller by either executing a *return* statement, or by reaching the end if it's body (it is important to note that this does not apply to `void` functions, since they do return to the caller, they just do not return any value). Such a function may end by calling `std::terminate` or `std::exit`, or by throwing an exception. It is also worth noting that such a function can return by executing `longjmp`.

For instance, the function below will always either throw an exception or call `std::terminate`, so it is a good candidate for `[[noreturn]]`:

```
[[noreturn]] void ownAssertFailureHandler(std::string message) {
    std::cerr << message << std::endl;
    if (THROW_EXCEPTION_ON_ASSERT)
        throw AssertException(std::move(message));
    std::terminate();
}
```

This kind of functionality allows the compiler to end a function without a return statement if it knows the code will never be executed. Here, because the call to `ownAssertFailureHandler` (defined above) in the code below will never return, the compiler does not need to add code below that call:

```
std::vector<int> createSequence(int end) {
    if (end > 0) {
        std::vector<int> sequence;
        sequence.reserve(end+1);
        for (int i = 0; i <= end; ++i)
            sequence.push_back(i);
        return sequence;
    }
    ownAssertFailureHandler("Negative number passed to createSequence()"s);
    // return std::vector<int>{}; //< Not needed because of [[noreturn]]
}
```

It is undefined behavior if the function will actually return, so the following is not allowed:

```
[[noreturn]] void assertPositive(int number) {
    if (number >= 0)
        return;
    else
```

```
        ownAssertFailureHandler("Positive number expected"s); //< [[noreturn]]
}
```

Note that the `[[noreturn]]` is mostly used in void functions. However, this is not a requirement, allowing the functions to be used in generic programming:

```
template<class InconsistencyHandler>
double fortyTwoDivideBy(int i) {
    if (i == 0)
         i = InconsistencyHandler::correct(i);
    return 42. / i;
}

struct InconsistencyThrower {
    static [[noreturn]] int correct(int i) { ownAssertFailureHandler("Unknown inconsistency"s); }
}

struct InconsistencyChangeToOne {
    static int correct(int i) { return 1; }
}

double fortyTwo = fortyTwoDivideBy<InconsistencyChangeToOne>(0);
double unreachable = fortyTwoDivideBy<InconsistencyThrower>(0);
```

The following standard library functions have this attribute:

- std::abort
- std::exit
- std::quick_exit
- std::unexpected
- std::terminate
- std::rethrow_exception
- std::throw_with_nested
- std::nested_exception::rethrow_nested

Chapter 124: Recursion in C++

Section 124.1: Using tail recursion and Fibonnaci-style recursion to solve the Fibonnaci sequence

The simple and most obvious way to use recursion to get the Nth term of the Fibonnaci sequence is this

```
int get_term_fib(int n)
{
  if (n == 0)
    return 0;
  if (n == 1)
    return 1;
  return get_term_fib(n - 1) + get_term_fib(n - 2);
}
```

However, this algorithm does not scale for higher terms: for bigger and bigger n, the number of function calls that you need to make grows exponentially. This can be replaced with a simple tail recursion.

```
int get_term_fib(int n, int prev = 0, int curr = 1)
{
  if (n == 0)
    return prev;
  if (n == 1)
    return curr;
  return get_term_fib(n - 1, curr, prev + curr);
}
```

Each call to the function now immediately calculates the next term in the Fibonnaci sequence, so the number of function calls scales linearly with n.

Section 124.2: Recursion with memoization

Recursive functions can get quite expensive. If they are pure functions (functions that always return the same value when called with the same arguments, and that neither depend on nor modify external state), they can be made considerably faster at the expense of memory by storing the values already calculated.

The following is an implementation of the Fibonacci sequence with memoization:

```
#include <map>

int fibonacci(int n)
{
  static std::map<int, int> values;
  if (n==0 || n==1)
    return n;
  std::map<int,int>::iterator iter = values.find(n);
  if (iter == values.end())
  {
    return values[n] = fibonacci(n-1) + fibonacci(n-2);
  }
  else
  {
    return iter->second;
  }
}
```

Note that despite using the simple recursion formula, on first call this function is $O(n)$. On subsequent calls with the same value, it is of course $O(1)$.

Note however that this implementation is not reentrant. Also, it doesn't allow to get rid of stored values. An alternative implementation would be to allow the map to be passed as additional argument:

```
#include <map>

int fibonacci(int n, std::map<int, int> values)
{
  if (n==0 || n==1)
    return n;
  std::map<int,int>::iterator iter = values.find(n);
  if (iter == values.end())
  {
    return values[n] = fibonacci(n-1) + fibonacci(n-2);
  }
  else
  {
    return iter->second;
  }
}
```

For this version, the caller is required to maintain the map with the stored values. This has the advantage that the function is now reentrant, and that the caller can remove values that are no longer needed, saving memory. It has the disadvantage that it breaks encapsulation; the caller can change the output by populating the map with incorrect values.

Chapter 125: Arithmitic Metaprogramming

These are example of using C++ template metaprogramming in processing arithmitic operations in compile time.

Section 125.1: Calculating power in O(log n)

This example shows an efficient way of calculating power using template metaprogramming.

```
template <int base, unsigned int exponent>
struct power
{
    static const int halfvalue = power<base, exponent / 2>::value;
    static const int value = halfvalue * halfvalue * power<base, exponent % 2>::value;
};

template <int base>
struct power<base, 0>
{
   static const int value = 1;
   static_assert(base != 0, "power<0, 0> is not allowed");
};


template <int base>
struct power<base, 1>
{
    static const int value = base;
};
```

Example Usage:

```
std::cout << power<2, 9>::value;
```

Version ≥ C++14

This one also handles negative exponents:

```
template <int base, int exponent>
struct powerDouble
{
    static const int exponentAbs = exponent < 0 ? (-exponent) : exponent;
    static const int halfvalue = powerDouble<base, exponentAbs / 2>::intermediateValue;
    static const int intermediateValue = halfvalue * halfvalue * powerDouble<base, exponentAbs %
2>::intermediateValue;

    constexpr static double value = exponent < 0 ? (1.0 / intermediateValue) : intermediateValue;

};

template <int base>
struct powerDouble<base, 0>
{
    static const int intermediateValue = 1;
    constexpr static double value = 1;
    static_assert(base != 0, "powerDouble<0, 0> is not allowed");
};


template <int base>
```

```
struct powerDouble<base, 1>
{
    static const int intermediateValue = base;
    constexpr static double value = base;
};


int main()
{
    std::cout << powerDouble<2,-3>::value;
}
```

Chapter 126: Callable Objects

Callable objects are the collection of all C++ structures which can be used as a function. In practice, this are all things you can pass to the C++17 STL function invoke() or which can be used in the constructor of std::function, this includes: Function pointers, Classes with operator(), Classes with implicit conversions, References to functions, Pointers to member functions, Pointers to member data, lambdas. The callable objects are used in many STL algorithms as predicate.

Section 126.1: Function Pointers

Function pointers are the most basic way of passing functions around, which can also be used in C. (See the C documentation for more details).

For the purpose of callable objects, a function pointer can be defined as:

```
typedef returnType(*name)(arguments);                       // All
using name = returnType(*)(arguments);                      // <= C++11
using name = std::add_pointer<returnType(arguments)>::type; // <= C++11
using name = std::add_pointer_t<returnType(arguments)>;     // <= C++14
```

If we would be using a function pointer for writing our own vector sort, it would look like:

```
using LessThanFunctionPtr = std::add_pointer_t<bool(int, int)>;
void sortVectorInt(std::vector<int>&v, LessThanFunctionPtr lessThan) {
    if (v.size() < 2)
        return;
    if (v.size() == 2) {
        if (!lessThan(v.front(), v.back())) // Invoke the function pointer
            std::swap(v.front(), v.back());
        return;
    }
    std::sort(v, lessThan);
}

bool lessThanInt(int lhs, int rhs) { return lhs < rhs; }
sortVectorInt(vectorOfInt, lessThanInt); // Passes the pointer to a free function

struct GreaterThanInt {
   static bool cmp(int lhs, int rhs) { return lhs > rhs; }
};
sortVectorInt(vectorOfInt, &GreaterThanInt::cmp); // Passes the pointer to a static member function
```

Alternatively, we could have invoked the function pointer one of following ways:

- `(*lessThan)(v.front(), v.back()) // All`
- `std::invoke(lessThan, v.front(), v.back()) // <= C++17`

Section 126.2: Classes with operator() (Functors)

Every class which overloads the `operator()` can be used as a function object. These classes can be written by hand (often referred to as functors) or automatically generated by the compiler by writing Lambdas from C++11 on.

```
struct Person {
    std::string name;
    unsigned int age;
};
```

```
// Functor which find a person by name
struct FindPersonByName {
    FindPersonByName(const std::string &name) : _name(name) {}

    // Overloaded method which will get called
    bool operator()(const Person &person) const {
         return person.name == _name;
    }
private:
    std::string _name;
};

std::vector<Person> v; // Assume this contains data
std::vector<Person>::iterator iFind =
    std::find_if(v.begin(), v.end(), FindPersonByName("Foobar"));
// ...
```

As functors have their own identity, they cannot be put in a typedef and these have to be accepted via template argument. The definition of `std::find_if` can look like:

```
template<typename Iterator, typename Predicate>
Iterator find_if(Iterator begin, Iterator end, Predicate &predicate) {
     for (Iterator i = begin, i != end, ++i)
         if (predicate(*i))
             return i;
     return end;
}
```

From C++17 on, the calling of the predicate can be done with invoke: `std::invoke(predicate, *i)`.

Chapter 127: Client server examples

Section 127.1: Hello TCP Client

This program is complimentary to Hello TCP Server program, you can run either of them to check the validity of each other. The program flow is quite common with Hello TCP server, so make sure to take a look at that too.

Here's the code -

```
#include <cstring>
#include <iostream>
#include <string>

#include <arpa/inet.h>
#include <netdb.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <unistd.h>


int main(int argc, char *argv[])
{
    // Now we're taking an ipaddress and a port number as arguments to our program
    if (argc != 3) {
        std::cerr << "Run program as 'program <ipaddress> <port>'\n";
        return -1;
    }


    auto &ipAddress = argv[1];
    auto &portNum   = argv[2];

    addrinfo hints, *p;
    memset(&hints, 0, sizeof(hints));
    hints.ai_family   = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_flags    = AI_PASSIVE;

    int gAddRes = getaddrinfo(ipAddress, portNum, &hints, &p);
    if (gAddRes != 0) {
        std::cerr << gai_strerror(gAddRes) << "\n";
        return -2;
    }

    if (p == NULL) {
        std::cerr << "No addresses found\n";
        return -3;
    }

    // socket() call creates a new socket and returns it's descriptor
    int sockFD = socket(p->ai_family, p->ai_socktype, p->ai_protocol);
    if (sockFD == -1) {
        std::cerr << "Error while creating socket\n";
        return -4;
    }

    // Note: there is no bind() call as there was in Hello TCP Server
    // why? well you could call it though it's not necessary
    // because client doesn't necessarily has to have a fixed port number
    // so next call will bind it to a random available port number
```

```
    // connect() call tries to establish a TCP connection to the specified server
    int connectR = connect(sockFD, p->ai_addr, p->ai_addrlen);
    if (connectR == -1) {
        close(sockFD);
        std::cerr << "Error while connecting socket\n";
        return -5;
    }

    std::string reply(15, ' ');

    // recv() call tries to get the response from server
    // BUT there's a catch here, the response might take multiple calls
    // to recv() before it is completely received
    // will be demonstrated in another example to keep this minimal
    auto bytes_recv = recv(sockFD, &reply.front(), reply.size(), 0);
    if (bytes_recv == -1) {
        std::cerr << "Error while receiving bytes\n";
        return -6;
    }

    std::cout << "\nClient recieved: " << reply << std::endl;
    close(sockFD);
    freeaddrinfo(p);

    return 0;
}
```

Section 127.2: Hello TCP Server

Let me start by saying you should first visit Beej's Guide to Network Programming and give it a quick read, which explains most of this stuff a bit more verbosely. We'll be creating a simple TCP server here which will say "Hello World" to all incoming connections and then close them. Another thing to note is, the server will be communicating to clients iteratively, which means one client at a time. Make sure to check out relevant man pages as they might contain valuable information about each function call and socket structures.

We'll run the server with a port, so we'll take an argument for port number as well. Let's get started with code -

```
#include <cstring>    // sizeof()
#include <iostream>
#include <string>

// headers for socket(), getaddrinfo() and friends
#include <arpa/inet.h>
#include <netdb.h>
#include <sys/socket.h>
#include <sys/types.h>

#include <unistd.h>    // close()


int main(int argc, char *argv[])
{
    // Let's check if port number is supplied or not..
    if (argc != 2) {
        std::cerr << "Run program as 'program <port>'\n";
        return -1;
    }

    auto &portNum = argv[1];
    const unsigned int backLog = 8;  // number of connections allowed on the incoming queue
```

```
    addrinfo hints, *res, *p;    // we need 2 pointers, res to hold and p to iterate over
    memset(&hints, 0, sizeof(hints));

    // for more explanation, man socket
    hints.ai_family   = AF_UNSPEC;    // don't specify which IP version to use yet
    hints.ai_socktype = SOCK_STREAM;  // SOCK_STREAM refers to TCP, SOCK_DGRAM will be?
    hints.ai_flags    = AI_PASSIVE;


    // man getaddrinfo
    int gAddRes = getaddrinfo(NULL, portNum, &hints, &res);
    if (gAddRes != 0) {
        std::cerr << gai_strerror(gAddRes) << "\n";
        return -2;
    }

    std::cout << "Detecting addresses" << std::endl;

    unsigned int numOfAddr = 0;
    char ipStr[INET6_ADDRSTRLEN];    // ipv6 length makes sure both ipv4/6 addresses can be stored
in this variable


    // Now since getaddrinfo() has given us a list of addresses
    // we're going to iterate over them and ask user to choose one
    // address for program to bind to
    for (p = res; p != NULL; p = p->ai_next) {
        void *addr;
        std::string ipVer;

        // if address is ipv4 address
        if (p->ai_family == AF_INET) {
            ipVer             = "IPv4";
            sockaddr_in *ipv4 = reinterpret_cast<sockaddr_in *>(p->ai_addr);
            addr              = &(ipv4->sin_addr);
            ++numOfAddr;
        }

        // if address is ipv6 address
        else {
            ipVer              = "IPv6";
            sockaddr_in6 *ipv6 = reinterpret_cast<sockaddr_in6 *>(p->ai_addr);
            addr               = &(ipv6->sin6_addr);
            ++numOfAddr;
        }

        // convert IPv4 and IPv6 addresses from binary to text form
        inet_ntop(p->ai_family, addr, ipStr, sizeof(ipStr));
        std::cout << "(" << numOfAddr << ") " << ipVer << " : " << ipStr
                  << std::endl;
    }

    // if no addresses found :(
    if (!numOfAddr) {
        std::cerr << "Found no host address to use\n";
        return -3;
    }

    // ask user to choose an address
    std::cout << "Enter the number of host address to bind with: ";
    unsigned int choice = 0;
    bool madeChoice     = false;
```

```
    do {
        std::cin >> choice;
        if (choice > (numOfAddr + 1) || choice < 1) {
            madeChoice = false;
            std::cout << "Wrong choice, try again!" << std::endl;
        } else
            madeChoice = true;
    } while (!madeChoice);



    p = res;

    // let's create a new socket, socketFD is returned as descriptor
    // man socket for more information
    // these calls usually return -1 as result of some error
    int sockFD = socket(p->ai_family, p->ai_socktype, p->ai_protocol);
    if (sockFD == -1) {
        std::cerr << "Error while creating socket\n";
        freeaddrinfo(res);
        return -4;
    }


    // Let's bind address to our socket we've just created
    int bindR = bind(sockFD, p->ai_addr, p->ai_addrlen);
    if (bindR == -1) {
        std::cerr << "Error while binding socket\n";

        // if some error occurs, make sure to close socket and free resources
        close(sockFD);
        freeaddrinfo(res);
        return -5;
    }


    // finally start listening for connections on our socket
    int listenR = listen(sockFD, backLog);
    if (listenR == -1) {
        std::cerr << "Error while Listening on socket\n";

        // if some error occurs, make sure to close socket and free resources
        close(sockFD);
        freeaddrinfo(res);
        return -6;
    }


    // structure large enough to hold client's address
    sockaddr_storage client_addr;
    socklen_t client_addr_size = sizeof(client_addr);


    const std::string response = "Hello World";


    // a fresh infinite loop to communicate with incoming connections
    // this will take client connections one at a time
    // in further examples, we're going to use fork() call for each client connection
    while (1) {

        // accept call will give us a new socket descriptor
```

```
        int newFD
          = accept(sockFD, (sockaddr *) &client_addr, &client_addr_size);
        if (newFD == -1) {
            std::cerr << "Error while Accepting on socket\n";
            continue;
        }

        // send call sends the data you specify as second param and it's length as 3rd param, also
returns how many bytes were actually sent
        auto bytes_sent = send(newFD, response.data(), response.length(), 0);
        close(newFD);
    }

    close(sockFD);
    freeaddrinfo(res);

    return 0;
}
```

The following program runs as -

```
Detecting addresses
(1) IPv4 : 0.0.0.0
(2) IPv6 : ::
Enter the number of host address to bind with: 1
```

Chapter 128: Const Correctness

Section 128.1: The Basics

`const` *correctness* is the practice of designing code so that only code that *needs* to modify an instance is *able* to modify an instance (i.e. has write access), and conversely, that any code that doesn't need to modify an instance is unable to do so (i.e. only has read access). This prevents the instance from being modified unintentionally, making code less errorprone, and documents whether the code is intended to change the instance's state or not. It also allows instances to be treated as `const` whenever they don't need to be modified, or defined as `const` if they don't need to be changed after initialisation, without losing any functionality.

This is done by giving member functions `const` CV-qualifiers, and by making pointer/reference parameters `const`, except in the case that they need write access.

```
class ConstCorrectClass {
    int x;

  public:
    int getX() const { return x; } // Function is const: Doesn't modify instance.
    void setX(int i) { x = i; }    // Not const: Modifies instance.
};

// Parameter is const: Doesn't modify parameter.
int const_correct_reader(const ConstCorrectClass& c) {
    return c.getX();
}

// Parameter isn't const: Modifies parameter.
void const_correct_writer(ConstCorrectClass& c) {
    c.setX(42);
}

const ConstCorrectClass invariant; // Instance is const: Can't be modified.
ConstCorrectClass         variant; // Instance isn't const: Can be modified.

// ...

const_correct_reader(invariant); // Good.   Calling non-modifying function on const instance.
const_correct_reader(variant);   // Good.   Calling non-modifying function on modifiable instance.

const_correct_writer(variant);   // Good.   Calling modifying function on modifiable instance.
const_correct_writer(invariant); // Error.  Calling modifying function on const instance.
```

Due to the nature of const correctness, this starts with the class' member functions, and works its way outwards; if you try to call a non-`const` member function from a `const` instance, or from a non-`const` instance being treated as `const`, the compiler will give you an error about it losing cv-qualifiers.

Section 128.2: Const Correct Class Design

In a `const`-correct class, all member functions which don't change logical state have `this` cv-qualified as `const`, indicating that they don't modify the object (apart from any `mutable` fields, which can freely be modified even in `const` instances); if a `const` cv-qualified function returns a reference, that reference should also be `const`. This allows them to be called on both constant and non-cv-qualified instances, as a `const T*` is capable of binding to either a `T*` or a `const T*`. This, in turn, allows functions to declare their passed-by-reference parameters as `const` when they don't need to be modified, without losing any functionality.

Furthermore, in a `const` correct class, all passed-by-reference function parameters will be `const` correct, as discussed in `Const Correct Function Parameters`, so that they can only be modified when the function explicitly *needs* to modify them.

First, let's look at `this` cv-qualifiers:

```
// Assume class Field, with member function "void insert_value(int);".

class ConstIncorrect {
    Field fld;

  public:
    ConstIncorrect(Field& f); // Modifies.

    Field& getField();        // Might modify.  Also exposes member as non-const reference,
                              //  allowing indirect modification.
    void setField(Field& f);  // Modifies.

    void doSomething(int i);  // Might modify.
    void doNothing();         // Might modify.
};

ConstIncorrect::ConstIncorrect(Field& f) : fld(f) {} // Modifies.
Field& ConstIncorrect::getField() { return fld; }    // Doesn't modify.
void ConstIncorrect::setField(Field& f) { fld = f; } // Modifies.
void ConstIncorrect::doSomething(int i) {            // Modifies.
    fld.insert_value(i);
}
void ConstIncorrect::doNothing() {}                  // Doesn't modify.


class ConstCorrectCVQ {
    Field fld;

  public:
    ConstCorrectCVQ(Field& f);     // Modifies.

    const Field& getField() const; // Doesn't modify.  Exposes member as const reference,
                                   //  preventing indirect modification.
    void setField(Field& f);       // Modifies.

    void doSomething(int i);       // Modifies.
    void doNothing() const;        // Doesn't modify.
};

ConstCorrectCVQ::ConstCorrectCVQ(Field& f) : fld(f) {}
Field& ConstCorrectCVQ::getField() const { return fld; }
void ConstCorrectCVQ::setField(Field& f) { fld = f; }
void ConstCorrectCVQ::doSomething(int i) {
    fld.insert_value(i);
}
void ConstCorrectCVQ::doNothing() const  {}

// This won't work.
// No member functions can be called on const ConstIncorrect instances.
void const_correct_func(const ConstIncorrect& c) {
    Field f = c.getField();
    c.do_nothing();
}

// But this will.
```

```
// getField() and doNothing() can be called on const ConstCorrectCVQ instances.
void const_correct_func(const ConstCorrectCVQ& c) {
    Field f = c.getField();
    c.do_nothing();
}
```

We can then combine this with `Const Correct Function Parameters`, causing the class to be fully `const`-correct.

```
class ConstCorrect {
    Field fld;

  public:
    ConstCorrect(const Field& f);  // Modifies instance.  Doesn't modify parameter.

    const Field& getField() const; // Doesn't modify.  Exposes member as const reference,
                                   //  preventing indirect modification.
    void setField(const Field& f); // Modifies instance.  Doesn't modify parameter.

    void doSomething(int i);       // Modifies.  Doesn't modify parameter (passed by value).
    void doNothing() const;        // Doesn't modify.
};

ConstCorrect::ConstCorrect(const Field& f) : fld(f) {}
Field& ConstCorrect::getField() const { return fld; }
void ConstCorrect::setField(const Field& f) { fld = f; }
void ConstCorrect::doSomething(int i) {
    fld.insert_value(i);
}
void ConstCorrect::doNothing() const {}
```

This can also be combined with overloading based on `const`ness, in the case that we want one behaviour if the instance is `const`, and a different behaviour if it isn't; a common use for this is constainers providing accessors that only allow modification if the container itself is non-`const`.

```
class ConstCorrectContainer {
    int arr[5];

  public:
    // Subscript operator provides read access if instance is const, or read/write access
    // otherwise.
          int& operator[](size_t index)       { return arr[index]; }
    const int& operator[](size_t index) const { return arr[index]; }

    // ...
};
```

This is commonly used in the standard library, with most containers providing overloads to take `const`ness into account.

Section 128.3: Const Correct Function Parameters

In a `const`-correct function, all passed-by-reference parameters are marked as `const` unless the function directly or indirectly modifies them, preventing the programmer from inadvertently changing something they didn't mean to change. This allows the function to take both `const` and non-cv-qualified instances, and in turn, causes the instance's `this` to be of type `const T*` when a member function is called, where `T` is the class' type.

```
struct Example {
    void func()       { std::cout << 3 << std::endl; }
```

```
    void func() const { std::cout << 5 << std::endl; }
};

void const_incorrect_function(Example& one, Example* two) {
    one.func();
    two->func();
}

void const_correct_function(const Example& one, const Example* two) {
    one.func();
    two->func();
}

int main() {
    Example a, b;
    const_incorrect_function(a, &b);
    const_correct_function(a, &b);
}

// Output:
3
3
5
5
```

While the effects of this are less immediately visible than those of `const` correct class design (in that `const`-correct functions and `const`-incorrect classes will cause compilation errors, while `const`-correct classes and `const`-incorrect functions will compile properly), `const` correct functions will catch a lot of errors that `const` incorrect functions would let slip through, such as the one below. [Note, however, that a `const`-incorrect function *will* cause compilation errors if passed a `const` instance when it expected a non-`const` one.]

```
// Read value from vector, then compute & return a value.
// Caches return values for speed.
template<typename T>
const T& bad_func(std::vector<T>& v, Helper<T>& h) {
    // Cache values, for future use.
    // Once a return value has been calculated, it's cached & its index is registered.
    static std::vector<T> vals = {};

    int v_ind = h.get_index();               // Current working index for v.
    int vals_ind = h.get_cache_index(v_ind); // Will be -1 if cache index isn't registered.

    if (vals.size() && (vals_ind != -1) && (vals_ind < vals.size()) && !(h.needs_recalc())) {
        return vals[h.get_cache_index(v_ind)];
    }

    T temp = v[v_ind];

    temp -= h.poll_device();
    temp *= h.obtain_random();
    temp += h.do_tedious_calculation(temp, v[h.get_last_handled_index()]);

    // We're feeling tired all of a sudden, and this happens.
    if (vals_ind != -1) {
        vals[vals_ind] = temp;
    } else {
        v.push_back(temp);  // Oops.  Should've been accessing vals.
        vals_ind = vals.size() - 1;
        h.register_index(v_ind, vals_ind);
    }
```

```
    return vals[vals_ind];
}

// Const correct version.  Is identical to above version, so most of it shall be skipped.
template<typename T>
const T& good_func(const std::vector<T>& v, Helper<T>& h) {
    // ...

    // We're feeling tired all of a sudden, and this happens.
    if (vals_ind != -1) {
        vals[vals_ind] = temp;
    } else {
        v.push_back(temp);  // Error: discards qualifiers.
        vals_ind = vals.size() - 1;
        h.register_index(v_ind, vals_ind);
    }

    return vals[vals_ind];
}
```

Section 128.4: Const Correctness as Documentation

One of the more useful things about `const` correctness is that it serves as a way of documenting code, providing certain guarantees to the programmer and other users. These guarantees are enforced by the compiler due to `const`ness, with a lack of `const`ness in turn indicating that code doesn't provide them.

`const` CV-Qualified Member Functions:

- Any member function which is `const` can be assumed to have intent to read the instance, and:
  - Shall not modify the logical state of the instance they are called on. Therefore, they shall not modify any member variables of the instance they are called on, except `mutable` variables.
  - Shall not call any *other* functions that would modify any member variables of the instance, except `mutable` variables.
- Conversely, any member function which isn't `const` can be assumed to have intent to modify the instance, and:
  - May or may not modify logical state.
  - May or may not call other functions which modify logical state.

This can be used to make assumptions about the state of the object after any given member function is called, even without seeing the definition of that function:

```
// ConstMemberFunctions.h

class ConstMemberFunctions {
    int val;
    mutable int cache;
    mutable bool state_changed;

  public:
    // Constructor clearly changes logical state.  No assumptions necessary.
    ConstMemberFunctions(int v = 0);

    // We can assume this function doesn't change logical state, and doesn't call
    //  set_val().  It may or may not call squared_calc() or bad_func().
    int calc() const;

    // We can assume this function doesn't change logical state, and doesn't call
    //  set_val().  It may or may not call calc() or bad_func().
```

```
    int squared_calc() const;

    // We can assume this function doesn't change logical state, and doesn't call
    //  set_val().  It may or may not call calc() or squared_calc().
    void bad_func() const;

    // We can assume this function changes logical state, and may or may not call
    //  calc(), squared_calc(), or bad_func().
    void set_val(int v);
};
```

Due to `const` rules, these assumptions will in fact be enforced by the compiler.

```
// ConstMemberFunctions.cpp

ConstMemberFunctions::ConstMemberFunctions(int v /* = 0*/)
  : cache(0), val(v), state_changed(true) {}

// Our assumption was correct.
int ConstMemberFunctions::calc() const {
    if (state_changed) {
        cache = 3 * val;
        state_changed = false;
    }

    return cache;
}

// Our assumption was correct.
int ConstMemberFunctions::squared_calc() const {
    return calc() * calc();
}

// Our assumption was incorrect.
// Function fails to compile, due to `this` losing qualifiers.
void ConstMemberFunctions::bad_func() const {
    set_val(863);
}

// Our assumption was correct.
void ConstMemberFunctions::set_val(int v) {
    if (v != val) {
        val = v;
        state_changed = true;
    }
}
```

`const` Function Parameters:

- Any function with one or more parameters which are `const` can be assumed to have intent to read those parameters, and:
    - Shall not modify those parameters, or call any member functions that would modify them.
    - Shall not pass those parameters to any *other* function which would modify them and/or call any member functions that would modify them.
- Conversely, any function with one or more parameters which aren't `const` can be assumed to have intent to modify those parameters, and:
    - May or may not modify those parameters, or call any member functions which whould modify them.
    - May or may not pass those parameters to other functions which would modify them and/or call any member functions that would modify them.

This can be used to make assumptions about the state of the parameters after being passed to any given function, even without seeing the definition of that function.

```
// function_parameter.h

// We can assume that c isn't modified (and c.set_val() isn't called), and isn't passed
//  to non_qualified_function_parameter().  If passed to one_const_one_not(), it is the first
//  parameter.
void const_function_parameter(const ConstMemberFunctions& c);

// We can assume that c is modified and/or c.set_val() is called, and may or may not be passed
//  to any of these functions.  If passed to one_const_one_not, it may be either parameter.
void non_qualified_function_parameter(ConstMemberFunctions& c);

// We can assume that:
  // l is not modified, and l.set_val() won't be called.
  // l may or may not be passed to const_function_parameter().
  // r is modified, and/or r.set_val() may be called.
  // r may or may not be passed to either of the preceding functions.
void one_const_one_not(const ConstMemberFunctions& l, ConstMemberFunctions& r);

// We can assume that c isn't modified (and c.set_val() isn't called), and isn't passed
//  to non_qualified_function_parameter().  If passed to one_const_one_not(), it is the first
//  parameter.
void bad_parameter(const ConstMemberFunctions& c);
```

Due to `const` rules, these assumptions will in fact be enforced by the compiler.

```
// function_parameter.cpp

// Our assumption was correct.
void const_function_parameter(const ConstMemberFunctions& c) {
    std::cout << "With the current value, the output is: " << c.calc() << '\n'
              << "If squared, it's: " << c.squared_calc()
              << std::endl;
}

// Our assumption was correct.
void non_qualified_function_parameter(ConstMemberFunctions& c) {
    c.set_val(42);
    std::cout << "For the value 42, the output is: " << c.calc() << '\n'
              << "If squared, it's: " << c.squared_calc()
              << std::endl;
}

// Our assumption was correct, in the ugliest possible way.
// Note that const correctness doesn't prevent encapsulation from intentionally being broken,
//  it merely prevents code from having write access when it doesn't need it.
void one_const_one_not(const ConstMemberFunctions& l, ConstMemberFunctions& r) {
    // Let's just punch access modifiers and common sense in the face here.
    struct Machiavelli {
        int val;
        int unimportant;
        bool state_changed;
    };
    reinterpret_cast<Machiavelli&>(r).val = l.calc();
    reinterpret_cast<Machiavelli&>(r).state_changed = true;

    const_function_parameter(l);
    const_function_parameter(r);
}
```

```
// Our assumption was incorrect.
// Function fails to compile, due to `this` losing qualifiers in c.set_val().
void bad_parameter(const ConstMemberFunctions& c) {
    c.set_val(18);
}
```

While it *is* possible to circumvent const correctness, and by extension break these guarantees, this must be done intentionally by the programmer (just like breaking encapsulation with Machiavelli, above), and is likely to cause undefined behaviour.

```
class DealBreaker : public ConstMemberFunctions {
  public:
    DealBreaker(int v = 0);

    // A foreboding name, but it's const...
    void no_guarantees() const;
}

DealBreaker::DealBreaker(int v /* = 0 */) : ConstMemberFunctions(v) {}

// Our assumption was incorrect.
// const_cast removes const-ness, making the compiler think we know what we're doing.
void DealBreaker::no_guarantees() const {
    const_cast<DealBreaker*>(this)->set_val(823);
}

// ...

const DealBreaker d(50);
d.no_guarantees(); // Undefined behaviour: d really IS const, it may or may not be modified.
```

However, due to this requiring the programmer to very specifically *tell* the compiler that they intend to ignore constness, and being inconsistent across compilers, it is generally safe to assume that const correct code will refrain from doing so unless otherwise specified.

Chapter 129: Parameter packs

Section 129.1: A template with a parameter pack

```
template<class ... Types> struct Tuple {};
```

A parameter pack is a template parameter accepting zero or more template arguments. If a template has at least one parameter pack is a *variadic template*.

Section 129.2: Expansion of a parameter pack

The pattern `parameter_pack ...` is expanded into a list of comma-separated substitutions of `parameter_pack` with each one of its parameters

```
template<class T> // Base of recursion
void variadic_printer(T last_argument) {
    std::cout << last_argument;
}

template<class T, class ...Args>
void variadic_printer(T first_argument, Args... other_arguments) {
  std::cout << first_argument << "\n";
  variadic_printer(other_arguments...); // Parameter pack expansion
}
```

The code above invoked with `variadic_printer(1, 2, 3, "hello");` prints

```
1
2
3
hello
```

Chapter 130: Build Systems

C++, like C, has a long and varied history regarding compilation workflows and build processes. Today, C++ has various popular build systems that are used to compile programs, sometimes for multiple platforms within one build system. Here, a few build systems will be reviewed and analyzed.

Section 130.1: Generating Build Environment with CMake

CMake generates build environments for nearly any compiler or IDE from a single project definition. The following examples will demonstrate how to add a CMake file to the cross-platform "Hello World" C++ code.

CMake files are always named "CMakeLists.txt" and should already exist in every project's root directory (and possibly in sub-directories too.) A basic CMakeLists.txt file looks like:

```
cmake_minimum_required(VERSION 2.4)

project(HelloWorld)

add_executable(HelloWorld main.cpp)
```

See it live on Coliru.

This file tells CMake the project name, what file version to expect, and instructions to generate an executable called "HelloWorld" that requires `main.cpp`.

Generate a build environment for your installed compiler/IDE from the command line:

```
> cmake .
```

Build the application with:

```
> cmake --build .
```

This generates the default build environment for the system, depending on the OS and installed tools. Keep source code clean from any build artifacts with use of "out-of-source" builds:

```
> mkdir build
> cd build
> cmake ..
> cmake --build .
```

CMake can also abstract the platform shell's basic commands from the previous example:

```
> cmake -E make_directory build
> cmake -E chdir build cmake ..
> cmake --build build
```

CMake includes generators for a number of common build tools and IDEs. To generate makefiles for Visual Studio's nmake:

```
> cmake -G "NMake Makefiles" ..
> nmake
```

Section 130.2: Compiling with GNU make

Introduction

The GNU Make (styled `make`) is a program dedicated to the automation of executing shell commands. GNU Make is one specific program that falls under the Make family. Make remains popular among Unix-like and POSIX-like operating systems, including those derived from the Linux kernel, Mac OS X, and BSD.

GNU Make is especially notable for being attached to the GNU Project, which is attached to the popular GNU/Linux operating system. GNU Make also has compatible versions running on various flavors of Windows and Mac OS X. It is also a very stable version with historical significance that remains popular. It is for these reasons that GNU Make is often taught alongside C and C++.

Basic rules

To compile with make, create a Makefile in your project directory. Your Makefile could be as simple as:

Makefile

```
# Set some variables to use in our command
# First, we set the compiler to be g++
CXX=g++

# Then, we say that we want to compile with g++'s recommended warnings and some extra ones.
CXXFLAGS=-Wall -Wextra -pedantic

# This will be the output file
EXE=app

SRCS=main.cpp

# When you call `make` at the command line, this "target" is called.
# The $(EXE) at the right says that the `all` target depends on the `$(EXE)` target.
# $(EXE) expands to be the content of the EXE variable
# Note: Because this is the first target, it becomes the default target if `make` is called without
target
all: $(EXE)

# This is equivalent to saying
# app: $(SRCS)
# $(SRCS) can be separated, which means that this target would depend on each file.
# Note that this target has a "method body": the part indented by a tab (not four spaces).
# When we build this target, make will execute the command, which is:
# g++ -Wall -Wextra -pedantic -o app main.cpp
# I.E. Compile main.cpp with warnings, and output to the file ./app
$(EXE): $(SRCS)
	@$(CXX) $(CXXFLAGS) -o $@ $(SRCS)

# This target should reverse the `all` target. If you call
# make with an argument, like `make clean`, the corresponding target
# gets called.
clean:
	@rm -f $(EXE)
```

> **NOTE: Make absolutely sure that the indentations are with a tab, not with four spaces. Otherwise, you'll get an error of** `Makefile:10: *** missing separator. Stop.`

To run this from the command-line, do the following:

```
$ cd ~/Path/to/project
$ make
$ ls
app  main.cpp  Makefile

$ ./app
Hello World!

$ make clean
$ ls
main.cpp  Makefile
```

Incremental builds

When you start having more files, make becomes more useful. What if you edited **a.cpp** but not **b.cpp**? Recompiling **b.cpp** would take more time.

With the following directory structure:

```
.
+-- src
|   +-- a.cpp
|   +-- a.hpp
|   +-- b.cpp
|   +-- b.hpp
+-- Makefile
```

This would be a good Makefile:

Makefile

```
CXX=g++
CXXFLAGS=-Wall -Wextra -pedantic
EXE=app

SRCS_GLOB=src/*.cpp
SRCS=$(wildcard $(SRCS_GLOB))
OBJS=$(SRCS:.cpp=.o)

all: $(EXE)

$(EXE): $(OBJS)
    @$(CXX) -o $@ $(OBJS)

depend: .depend

.depend: $(SRCS)
    @-rm -f ./.depend
    @$(CXX) $(CXXFLAGS) -MM $^>>./.depend

clean:
    -rm -f $(EXE)
    -rm $(OBJS)
    -rm *~
    -rm .depend

include .depend
```

Again watch the tabs. This new Makefile ensures that you only recompile changed files, minimizing compile time.

Documentation

For more on make, see the official documentation by the Free Software Foundation, the stackoverflow documentation and dmckee's elaborate answer on stackoverflow.

Section 130.3: Building with SCons

You can build the cross-platform "Hello World" C++ code, using Scons - A Python-language software construction tool.

First, create a file called `SConstruct` (note that SCons will look for a file with this exact name by default). For now, the file should be in a directory right along your `hello.cpp`. Write in the new file the line

```
Program('hello.cpp')
```

Now, from the terminal, run `scons`. You should see something like

```
$ scons
scons: Reading SConscript files ...
scons: done reading SConscript files.
scons: Building targets ...
g++ -o hello.o -c hello.cpp
g++ -o hello hello.o
scons: done building targets.
```

(although the details will vary depending on your operating system and installed compiler).

The `Environment` and `Glob` classes will help you further configure what to build. E.g., the `SConstruct` file

```
env=Environment(CPPPATH='/usr/include/boost/',
    CPPDEFINES=[],
    LIBS=[],
    SCONS_CXX_STANDARD="c++11"
    )

env.Program('hello', Glob('src/*.cpp'))
```

builds the executable `hello`, using all `cpp` files in `src`. Its `CPPPATH` is `/usr/include/boost` and it specifies the C++11 standard.

Section 130.4: Autotools (GNU)

Introduction

The Autotools are a group of programs that create a GNU Build System for a given software package. It is a suite of tools that work together to produce various build resources, such as a Makefile (to be used with GNU Make). Thus, Autotools can be considered a de facto build system generator.

Some notable Autotools programs include:

- Autoconf
- Automake (not to be confused with `make`)

In general, Autotools is meant to generate the Unix-compatible script and Makefile to allow the following command

to build (as well as install) most packages (in the simple case):

```
./configure && make && make install
```

As such, Autotools also has a relationship with certain package managers, especially those that are attached to operating systems that conform to the POSIX Standard(s).

Section 130.5: Ninja

Introduction

The Ninja build system is described by its project website as "a small build system with a focus on speed." Ninja is designed to have its files generated by build system file generators, and takes a low-level approach to build systems, in contrast to higher-level build system managers like CMake or Meson.

Ninja is primarily written in C++ and Python, and was created as an alternative to the SCons build system for the Chromium project.

Section 130.6: NMAKE (Microsoft Program Maintenance Utility)

Introduction

NMAKE is a command-line utility developed by Microsoft to be used primarily in conjunction with Microsoft Visual Studio and/or the Visual C++ command line tools.

NMAKE is build system that falls under the Make family of build systems, but has certain distinct features that diverge from Unix-like Make programs, such as supporting Windows-specific file path syntax (which itself differs from Unix-style file paths).

Chapter 131: Concurrency With OpenMP

This topic covers the basics of concurrency in C++ using OpenMP. OpenMP is documented in more detail in the OpenMP tag.

Parallelism or concurrency implies the execution of code at the same time.

Section 131.1: OpenMP: Parallel Sections

This example illustrates the basics of executing sections of code in parallel.

As OpenMP is a built-in compiler feature, it works on any supported compilers without including any libraries. You may wish to include `omp.h` if you want to use any of the openMP API features.

Sample Code

```
std::cout << "begin ";
//    This pragma statement hints the compiler that the
//    contents within the { } are to be executed in as
//    parallel sections using openMP, the compiler will
//    generate this chunk of code for parallel execution
#pragma omp parallel sections
{
    //    This pragma statement hints the compiler that
    //    this is a section that can be executed in parallel
    //    with other section, a single section will be executed
    //    by a single thread.
    //    Note that it is "section" as opposed to "sections" above
    #pragma omp section
    {
        std::cout << "hello " << std::endl;
        /** Do something **/
    }
    #pragma omp section
    {
        std::cout << "world " << std::endl;
        /** Do something **/
    }
}
//    This line will not be executed until all the
//    sections defined above terminates
std::cout << "end" << std::endl;
```

Outputs

This example produces 2 possible outputs and is dependent on the operating system and hardware. The output also illustrates a **race condition** problem that would occur from such an implementation.

OUTPUT A	OUTPUT B
begin hello world end	begin world hello end

Section 131.2: OpenMP: Parallel Sections

This example shows how to execute chunks of code in parallel

```
std::cout << "begin ";
//    Start of parallel sections
```

```
#pragma omp parallel sections
{
    //    Execute these sections in parallel
    #pragma omp section
    {
        ... do something ...
        std::cout << "hello ";
    }
    #pragma omp section
    {
        ... do something ...
        std::cout << "world ";
    }
    #pragma omp section
    {
        ... do something ...
        std::cout << "forever ";
    }
}
//    end of parallel sections
std::cout << "end";
```

Output

- begin hello world forever end
- begin world hello forever end
- begin hello forever world end
- begin forever hello world end

As execution order is not guaranteed, you may observe any of the above output.

Section 131.3: OpenMP: Parallel For Loop

This example shows how to divide a loop into equal parts and execute them in parallel.

```
//    Splits element vector into element.size() / Thread Qty
//    and allocate that range for each thread.
#pragma omp parallel for
for    (size_t i = 0; i < element.size(); ++i)
    element[i] = ...

//    Example Allocation (100 element per thread)
//    Thread 1 : 0 ~ 99
//    Thread 2 : 100 ~ 199
//    Thread 2 : 200 ~ 299
//    ...

//    Continue process
//    Only when all threads completed their allocated
//    loop job
...
```

*Please take extra care to not modify the size of the vector used in parallel for loops as **allocated range indices doesn't update automatically**.

Section 131.4: OpenMP: Parallel Gathering / Reduction

This example illustrates a concept to perform reduction or gathering using `std::vector` and OpenMP.

Supposed we have a scenario where we want multiple threads to help us generate a bunch of stuff, `int` is used here for simplicity and can be replaced with other data types.

This is particularly useful when you need to merge results from slaves to avoid segement faults or memory access violations and do not wish to use libraries or custom sync container libraries.

```
//    The Master vector
//    We want a vector of results gathered from slave threads
std::vector<int> Master;

//    Hint the compiler to parallelize this { } of code
//    with all available threads (usually the same as logical processor qty)
#pragma omp parallel
{
    //    In this area, you can write any code you want for each
    //    slave thread, in this case a vector to hold each of their results
    //    We don't have to worry about how many threads were spawn or if we need
    //    to repeat this declaration or not.
    std::vector<int> Slave;

    //    Tell the compiler to use all threads allocated for this parallel region
    //    to perform this loop in parts. Actual load appx = 1000000 / Thread Qty
    //    The nowait keyword tells the compiler that the slave threads don't
    //    have to wait for all other slaves to finish this for loop job
    #pragma omp for nowait
    for (size_t i = 0; i < 1000000; ++i
    {
        /* Do something */
        ....
        Slave.push_back(...);
    }

    //    Slaves that finished their part of the job
    //    will perform this thread by thread one at a time
    //    critical section ensures that only 0 or 1 thread performs
    //    the { } at any time
    #pragma omp critical
    {
        //    Merge slave into master
        //    use move iterators instead, avoid copy unless
        //    you want to use it for something else after this section
        Master.insert(Master.end(),
                      std::make_move_iterator(Slave.begin()),
                      std::make_move_iterator(Slave.end()));
    }
}

//    Have fun with Master vector
...
```

Chapter 132: Resource Management

One of the hardest things to do in C and C++ is resource management. Thankfully, in C++, we have many ways to go about designing resource management in our programs. This article hopes to explain some of the idioms and methods used to manage allocated resources.

Section 132.1: Resource Acquisition Is Initialization

Resource Acquisition Is Initialization (RAII) is a common idiom in resource management. In the case of dynamic memory, it uses smart pointers to accomplish resource management. When using RAII, an acquired resource is immediately given ownership to a smart pointer or equivalent resource manager. The resource is only accessed through this manager, so the manager can keep track of various operations. For example, `std::auto_ptr` automatically frees its corresponding resource when it falls out of scope or is otherwise deleted.

```
#include <memory>
#include <iostream>
using namespace std;

int main() {
    {
        auto_ptr ap(new int(5)); // dynamic memory is the resource
        cout << *ap << endl; // prints 5
    } // auto_ptr is destroyed, its resource is automatically freed
}
```

Version ≥ C++11

`std::auto_ptr`'s main problem is that it can't copied without transferring ownership:

```
#include <memory>
#include <iostream>
using namespace std;

int main() {
    auto_ptr ap1(new int(5));
    cout << *ap1 << endl; // prints 5
    auto_ptr ap2(ap1); // copy ap2 from ap1; ownership now transfers to ap2
    cout << *ap2 << endl; // prints 5
    cout << ap1 == nullptr << endl; // prints 1; ap1 has lost ownership of resource
}
```

Because of these weird copy semantics, `std::auto_ptr` can't be used in containers, among other things. The reason it does this is to prevent deleting memory twice: if there are two `auto_ptrs` with ownership of the same resource, they both try to free it when they're destroyed. Freeing an already freed resource can generally cause problems, so it is important to prevent it. However, `std::shared_ptr` has a method to avoid this while not transferring ownership when copying:

```
#include <memory>
#include <iostream>
using namespace std;

int main() {
    shared_ptr sp2;
    {
        shared_ptr sp1(new int(5)); // give ownership to sp1
        cout << *sp1 << endl; // prints 5
        sp2 = sp1; // copy sp2 from sp1; both have ownership of resource
```

```
        cout << *sp1 << endl; // prints 5
        cout << *sp2 << endl; // prints 5
    } // sp1 goes out of scope and is destroyed; sp2 has sole ownership of resource
    cout << *sp2 << endl;
} // sp2 goes out of scope; nothing has ownership, so resource is freed
```

Section 132.2: Mutexes & Thread Safety

Problems may happen when multiple threads try to access a resource. For a simple example, suppose we have a thread that adds one to a variable. It does this by first reading the variable, adding one to it, then storing it back. Suppose we initialize this variable to 1, then create two instances of this thread. After both threads finish, intuition suggests that this variable should have a value of 3. However, the below table illustrates what might go wrong:

	Thread 1	**Thread 2**
Time Step 1	Read 1 from variable	
Time Step 2		Read 1 from variable
Time Step 3	Add 1 plus 1 to get 2	
Time Step 4		Add 1 plus 1 to get 2
Time Step 5	Store 2 into variable	
Time Step 6		Store 2 into variable

As you can see, at the end of the operation, 2 is in the variable, instead of 3. The reason is that Thread 2 read the variable before Thread 1 was done updating it. The solution? Mutexes.

A mutex (portmanteau of **mut**ual **ex**clusion) is a resource management object designed to solve this type of problem. When a thread wants to access a resource, it "acquires" the resource's mutex. Once it is done accessing the resource, the thread "releases" the mutex. While the mutex is acquired, all calls to acquire the mutex will not return until the mutex is released. To better understand this, think of a mutex as a waiting line at the supermarket: the threads go into line by trying to acquire the mutex and then waiting for the threads ahead of them to finish up, then using the resource, then stepping out of line by releasing the mutex. There would be complete pandemonium if everybody tried to access the resource at once.

Version ≥ C++11

`std::mutex` is C++11's implementation of a mutex.

```
#include <thread>
#include <mutex>
#include <iostream>
using namespace std;

void add_1(int& i, const mutex& m) { // function to be run in thread
    m.lock();
    i += 1;
    m.unlock();
}

int main() {
    int var = 1;
    mutex m;

    cout << var << endl; // prints 1

    thread t1(add_1, var, m); // create thread with arguments
    thread t2(add_1, var, m); // create another thread
    t1.join(); t2.join(); // wait for both threads to finish
```

```
    cout << var << endl; // prints 3
}
```

Chapter 133: Storage class specifiers

Storage class specifiers are keywords that can be used in declarations. They do not affect the type of the declaration, but typically modify the way in which the entity is stored.

Section 133.1: extern

The `extern` storage class specifier can modify a declaration in one of the three following ways, depending on context:

1. It can be used to declare a variable without defining it. Typically, this is used in a header file for a variable that will be defined in a separate implementation file.

```
// global scope
int x;             // definition; x will be default-initialized
extern int y;      // declaration; y is defined elsewhere, most likely another TU
extern int z = 42; // definition; "extern" has no effect here (compiler may warn)
```

2. It gives external linkage to a variable at namespace scope even if `const` or `constexpr` would have otherwise caused it to have internal linkage.

```
// global scope
const int w = 42;            // internal linkage in C++; external linkage in C
static const int x = 42;     // internal linkage in both C++ and C
extern const int y = 42;     // external linkage in both C++ and C
namespace {
    extern const int z = 42; // however, this has internal linkage since
                             // it's in an unnamed namespace
}
```

3. It redeclares a variable at block scope if it was previously declared with linkage. Otherwise, it declares a new variable with linkage, which is a member of the nearest enclosing namespace.

```
// global scope
namespace {
    int x = 1;
    struct C {
        int x = 2;
        void f() {
            extern int x;           // redeclares namespace-scope x
            std::cout << x << '\n'; // therefore, this prints 1, not 2
        }
    };
}
void g() {
    extern int y; // y has external linkage; refers to global y defined elsewhere
}
```

A function can also be declared `extern`, but this has no effect. It is usually used as a hint to the reader that a function declared here is defined in another translation unit. For example:

```
void f();        // typically a forward declaration; f defined later in this TU
extern void g(); // typically not a forward declaration; g defined in another TU
```

In the above code, if `f` were changed to `extern` and `g` to non-`extern`, it would not affect the correctness or semantics of the program at all, but would likely confuse the reader of the code.

Section 133.2: register

Version < C++17

A storage class specifier that hints to the compiler that a variable will be heavily used. The word "register" is related to the fact that a compiler might choose to store such a variable in a CPU register so that it can be accessed in fewer clock cycles. It was deprecated starting in C++11.

```
register int i = 0;
while (i < 100) {
    f(i);
    int g = i*i;
    i += h(i, g);
}
```

Both local variables and function parameters may be declared `register`. Unlike C, C++ does not place any restrictions on what can be done with a `register` variable. For example, it is valid to take the address of a `register` variable, but this may prevent the compiler from actually storing such a variable in a register.

Version ≥ C++17

The keyword `register` is unused and reserved. A program that uses the keyword `register` is ill-formed.

Section 133.3: static

The `static` storage class specifier has three different meanings.

1. Gives internal linkage to a variable or function declared at namespace scope.

```
// internal function; can't be linked to
static double semiperimeter(double a, double b, double c) {
    return (a + b + c)/2.0;
}
// exported to client
double area(double a, double b, double c) {
    const double s = semiperimeter(a, b, c);
    return sqrt(s*(s-a)*(s-b)*(s-c));
}
```

2. Declares a variable to have static storage duration (unless it is `thread_local`). Namespace-scope variables are implicitly static. A static local variable is initialized only once, the first time control passes through its definition, and is not destroyed every time its scope is exited.

```
void f() {
    static int count = 0;
    std::cout << "f has been called " << ++count << " times so far\n";
}
```

3. When applied to the declaration of a class member, declares that member to be a static member.

```
struct S {
    static S* create() {
```

```
        return new S;
    }
};
int main() {
    S* s = S::create();
}
```

Note that in the case of a static data member of a class, both 2 and 3 apply simultaneously: the `static` keyword both makes the member into a static data member and makes it into a variable with static storage duration.

Section 133.4: auto

Version ≤ C++03

Declares a variable to have automatic storage duration. It is redundant, since automatic storage duration is already the default at block scope, and the auto specifier is not allowed at namespace scope.

```
void f() {
    auto int x; // equivalent to: int x;
    auto y;     // illegal in C++; legal in C89
}
auto int z;     // illegal: namespace-scope variable cannot be automatic
```

In C++11, `auto` changed meaning completely, and is no longer a storage class specifier, but is instead used for type deduction.

Section 133.5: mutable

A specifier that can be applied to the declaration of a non-static, non-reference data member of a class. A mutable member of a class is not `const` even when the object is `const`.

```
class C {
    int x;
    mutable int times_accessed;
  public:
    C(): x(0), times_accessed(0) {
    }
    int get_x() const {
        ++times_accessed; // ok: const member function can modify mutable data member
        return x;
    }
    void set_x(int x) {
        ++times_accessed;
        this->x = x;
    }
};
```

Version ≥ C++11

A second meaning for `mutable` was added in C++11. When it follows the parameter list of a lambda, it suppresses the implicit `const` on the lambda's function call operator. Therefore, a mutable lambda can modify the values of entities captured by copy. See mutable lambdas for more details.

```
std::vector<int> my_iota(int start, int count) {
    std::vector<int> result(count);
    std::generate(result.begin(), result.end(),
                  [start]() mutable { return start++; });
    return result;
```

```
}
```

Note that `mutable` is *not* a storage class specifier when used this way to form a mutable lambda.

Chapter 134: Linkage specifications

A linkage specification tells the compiler to compile declarations in a way that allows them to be linked together with declarations written in another language, such as C.

Section 134.1: Signal handler for Unix-like operating system

Since a signal handler will be called by the kernel using the C calling convention, we must tell the compiler to use the C calling convention when compiling the function.

```
volatile sig_atomic_t death_signal = 0;
extern "C" void cleanup(int signum) {
    death_signal = signum;
}
int main() {
    bind(...);
    listen(...);
    signal(SIGTERM, cleanup);
    while (int fd = accept(...)) {
        if (fd == -1 && errno == EINTR && death_signal) {
            printf("Caught signal %d; shutting down\n", death_signal);
            break;
        }
        // ...
    }
}
```

Section 134.2: Making a C library header compatible with C++

A C library header can usually be included into a C++ program, since most declarations are valid in both C and C++. For example, consider the following `foo.h`:

```
typedef struct Foo {
    int bar;
} Foo;
Foo make_foo(int);
```

The definition of `make_foo` is separately compiled and distributed with the header in object form.

A C++ program can `#include <foo.h>`, but the compiler will not know that the `make_foo` function is defined as a C symbol, and will probably try to look for it with a mangled name, and fail to locate it. Even if it can find the definition of `make_foo` in the library, not all platforms use the same calling conventions for C and C++, and the C++ compiler will use the C++ calling convention when calling `make_foo`, which is likely to cause a segmentation fault if `make_foo` is expecting to be called with the C calling convention.

The way to remedy this problem is to wrap almost all the declarations in the header in an `extern "C"` block.

```
#ifdef __cplusplus
extern "C" {
#endif

typedef struct Foo {
    int bar;
} Foo;
Foo make_foo(int);

```

```
#ifdef __cplusplus
}  /* end of "extern C" block */
#endif
```

Now when `foo.h` is included from a C program, it will just appear as ordinary declarations, but when `foo.h` is included from a C++ program, `make_foo` will be inside an `extern "C"` block and the compiler will know to look for an unmangled name and use the C calling convention.

Chapter 135: Digit separators

Section 135.1: Digit Separator

Numeric literals of more than a few digits are hard to read.

- Pronounce 7237498123.
- Compare 237498123 with 237499123 for equality.
- Decide whether 237499123 or 20249472 is larger.

`C++14` define Simple Quotation Mark `'` as a digit separator, in numbers and user-defined literals. This can make it easier for human readers to parse large numbers.

Version ≥ C++14

```
long long decn = 1'000'000'000ll;
long long hexn = 0xFFFF'FFFFll;
long long octn = 00'23'00ll;
long long binn = 0b1010'0011ll;
```

Single quotes mark are ignored when determining its value.

Example:

- The literals `1048576`, `1'048'576`, `0X100000`, `0x10'0000`, and `0'004'000'000` all have the same value.
- The literals `1.602'176'565e-19` and `1.602176565e-19` have the same value.

The position of the single quotes is irrelevant. All the following are equivalent:

Version ≥ C++14

```
long long a1 = 123456789ll;
long long a2 = 123'456'789ll;
long long a3 = 12'34'56'78'9ll;
long long a4 = 12345'6789ll;
```

It is also allowed in `user-defined` literals:

Version ≥ C++14

```
std::chrono::seconds tiempo = 1'674'456s + 5'300h;
```

Chapter 136: C incompatibilities

This describes what C code will break in a C++ compiler.

Section 136.1: Reserved Keywords

The first example are keywords that have a special purpose in C++: the following is legal in C, but not C++.

```
int class = 5
```

These errors are easy to fix: just rename the variable.

Section 136.2: Weakly typed pointers

In C, pointers can be cast to a `void*`, which needs an explicit cast in C++. The following is illegal in C++, but legal in C:

```
void* ptr;
int* intptr = ptr;
```

Adding an explicit cast makes this work, but can cause further issues.

Section 136.3: goto or switch

In C++, you may not skip initializations with `goto` or `switch`. The following is valid in C, but not C++:

```
goto foo;
int skipped = 1;
foo;
```

These bugs may require redesign.

Chapter 137: Side by Side Comparisons of classic C++ examples solved via C++ vs C++11 vs C++14 vs C++17

Section 137.1: Looping through a container

In C++, looping through a sequence container `c` can be done using indexes as follows:

```
for(size_t i = 0; i < c.size(); ++i) c[i] = 0;
```

While simple, such writings are subject to common semantic errors, like wrong comparison operator, or wrong indexing variable:

```
for(size_t i = 0; i <= c.size(); ++j) c[i] = 0;
                     ^~~~~~~~~~~~~~^
```

Looping can also be achieved for all containers using iterators, with similar drawbacks:

```
for(iterator it = c.begin(); it != c.end(); ++it) (*it) = 0;
```

C++11 introduced range-based for loops and `auto` keyword, allowing the code to become:

```
for(auto& x : c) x = 0;
```

Here the only parameters are the container `c`, and a variable `x` to hold the current value. This prevents the semantics errors previously pointed.

According to the C++11 standard, the underlying implementation is equivalent to:

```
for(auto begin = c.begin(), end = c.end(); begin != end; ++begin)
{
    // ...
}
```

In such implementation, the expression `auto begin = c.begin(), end = c.end();` forces `begin` and `end` to be of the same type, while `end` is never incremented, nor dereferenced. So the range-based for loop only works for containers defined by a pair iterator/iterator. The C++17 standard relaxes this constraint by changing the implementation to:

```
auto begin = c.begin();
auto end = c.end();
for(; begin != end; ++begin)
{
    // ...
}
```

Here `begin` and `end` are allowed to be of different types, as long as they can be compared for inequality. This allows to loop through more containers, e.g. a container defined by a pair iterator/sentinel.

Chapter 138: Compiling and Building

Programs written in C++ need to be compiled before they can be run. There is a large variety of compilers available depending on your operating system.

Section 138.1: Compiling with GCC

Assuming a single source file named `main.cpp`, the command to compile and link an non-optimized executable is as follows (Compiling without optimization is useful for initial development and debugging, although `-Og` is officially recommended for newer GCC versions).

```
g++ -o app -Wall main.cpp -O0
```

To produce an optimized executable for use in production, use one of the `-O` options (see: `-O1`, `-O2`, `-O3`, `-Os`, `-Ofast`):

```
g++ -o app -Wall -O2 main.cpp
```

If the -O option is omitted, -O0, which means no optimizations, is used as default (specifying -O without a number resolves to -O1).

Alternatively, use optimization flags from the `O` groups (or more experimental optimizations) directly. The following example builds with `-O2` optimization, plus one flag from the `-O3` optimization level:

```
g++ -o app -Wall -O2 -ftree-partial-pre main.cpp
```

To produce a platform-specific optimized executable (for use in production on the machine with the same architecture), use:

```
g++ -o app -Wall -O2 -march=native main.cpp
```

Either of the above will produce a binary file that can be run with `.\app.exe` on Windows and `./app` on Linux, Mac OS, etc.

The `-o` flag can also be skipped. In this case, GCC will create default output executable `a.exe` on Windows and `a.out` on Unix-like systems. To compile a file without linking it, use the `-c` option:

```
g++ -o file.o -Wall -c file.cpp
```

This produces an object file named `file.o` which can later be linked with other files to produce a binary:

```
g++ -o app file.o otherfile.o
```

More about optimization options can be found at gcc.gnu.org. Of particular note are `-Og` (optimization with an emphasis on debugging experience -- recommended for the standard edit-compile-debug cycle) and `-Ofast` (all optimizations, including ones disregarding strict standards compliance).

The `-Wall` flag enables warnings for many common errors and should always be used. To improve code quality it is often encouraged also to use `-Wextra` and other warning flags which are not automatically enabled by `-Wall` and `-Wextra`.

If the code expects a specific C++ standard, specify which standard to use by including the `-std=` flag. Supported

values correspond to the year of finalization for each version of the ISO C++ standard. As of GCC 6.1.0, valid values for the `std=` flag are `c++98`/`c++03`, `c++11`, `c++14`, and `c++17`/`c++1z`. Values separated by a forward slash are equivalent.

```
g++ -std=c++11 <file>
```

GCC includes some compiler-specific extensions that are disabled when they conflict with a standard specified by the `-std=` flag. To compile with all extensions enabled, the value `gnu++XX` may be used, where `XX` is any of the years used by the `c++` values listed above.

The default standard will be used if none is specified. For versions of GCC prior to 6.1.0, the default is `-std=gnu++03`; in GCC 6.1.0 and greater, the default is `-std=gnu++14`.

Note that due to bugs in GCC, the `-pthread` flag must be present at compilation and linking for GCC to support the C++ standard threading functionality introduced with C++11, such as `std::thread` and `std::wait_for`. Omitting it when using threading functions may result in no warnings but invalid results on some platforms.

Linking with libraries:

Use the `-l` option to pass the library name:

```
g++ main.cpp -lpcre2-8
#pcre2-8 is the PCRE2 library for 8bit code units (UTF-8)
```

If the library is not in the standard library path, add the path with `-L` option:

```
g++ main.cpp -L/my/custom/path/ -lmylib
```

Multiple libraries can be linked together:

```
g++ main.cpp -lmylib1 -lmylib2 -lmylib3
```

If one library depends on another, put the dependent library **before** the independent library:

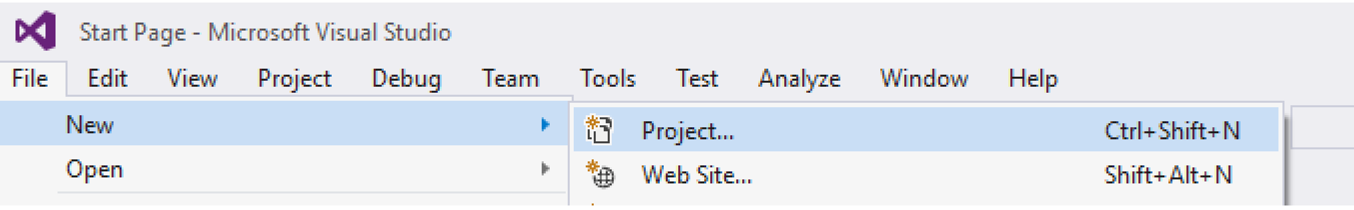
```
g++ main.cpp -lchild-lib -lbase-lib
```

Or let the linker determine the ordering itself via `--start-group` and `--end-group` (note: this has significant performance cost):

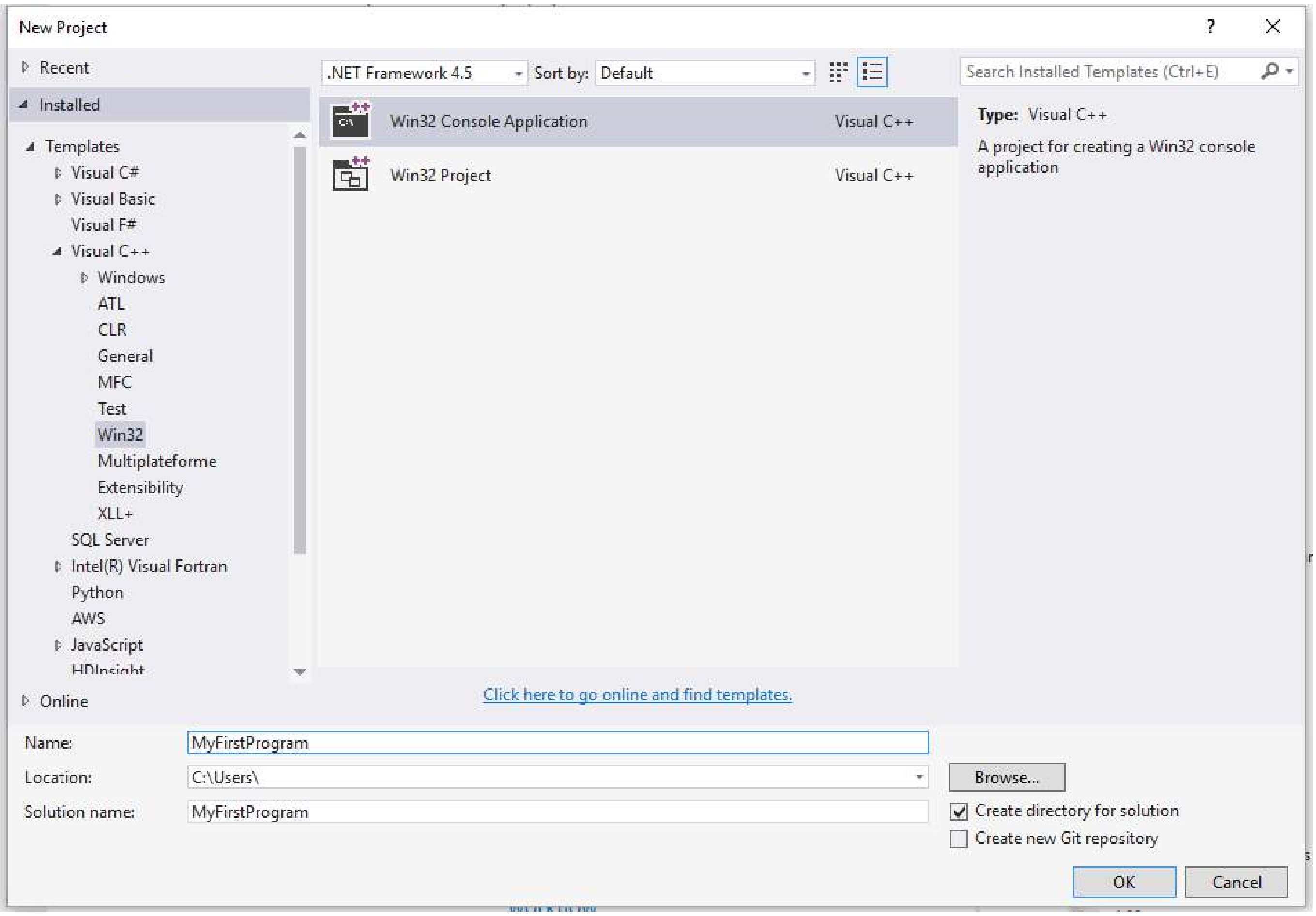
```
g++ main.cpp -Wl,--start-group -lbase-lib -lchild-lib -Wl,--end-group
```

Section 138.2: Compiling with Visual Studio (Graphical Interface) - Hello World

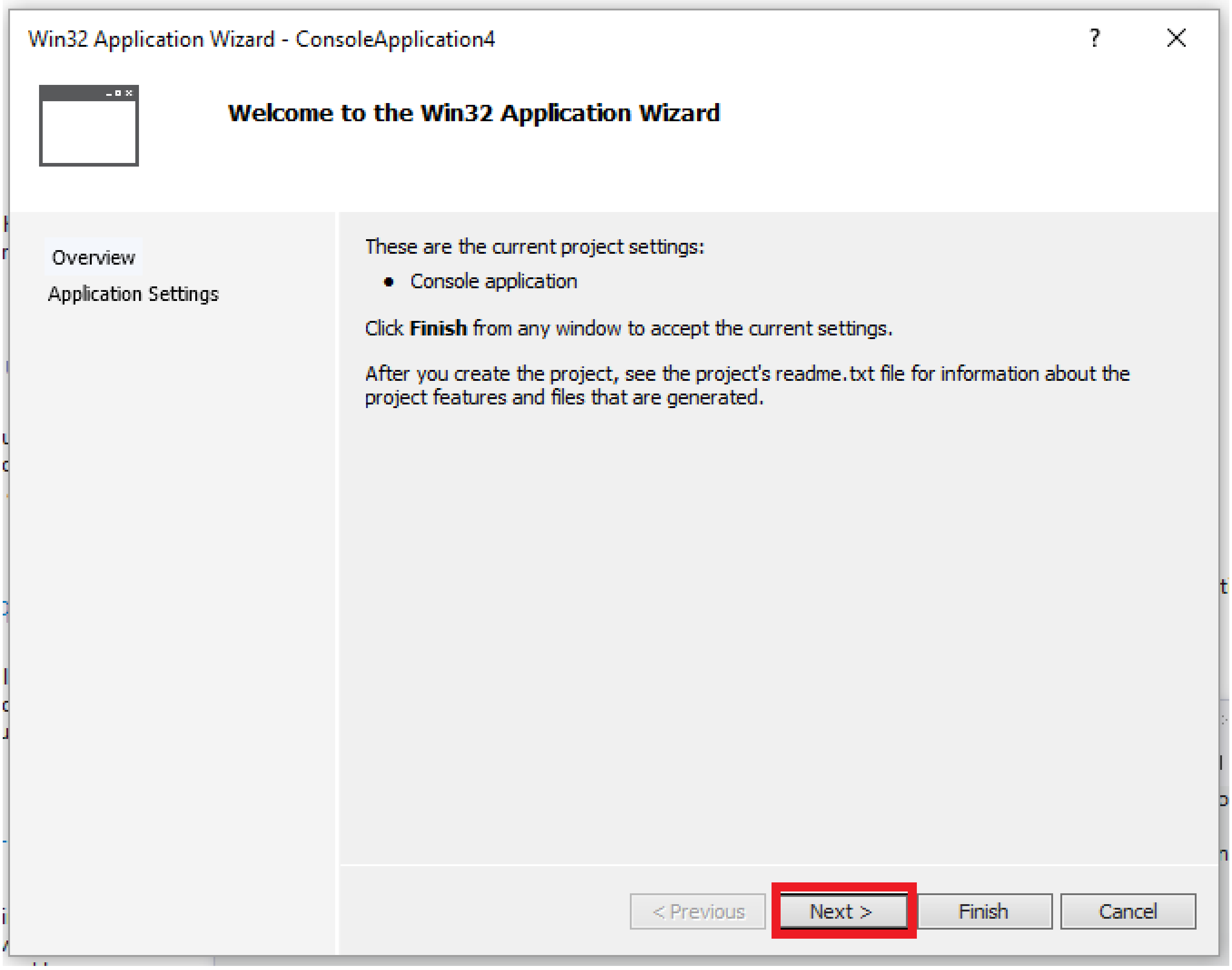
1. Download and install Visual Studio Community 2015
2. Open Visual Studio Community
3. Click File -> New -> Project



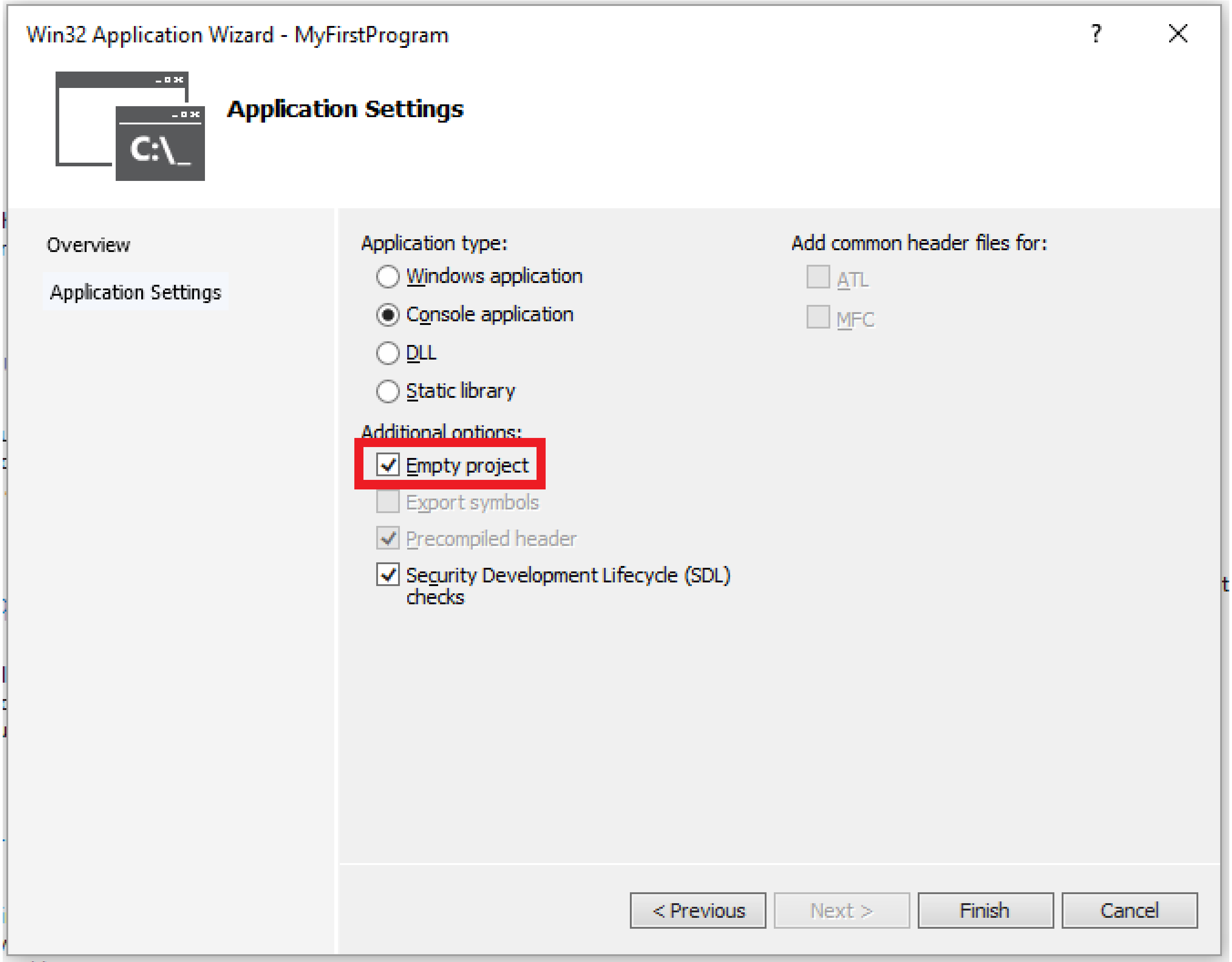
4. Click Templates -> Visual C++ -> Win32 Console Application and then name the project **MyFirstProgram**.



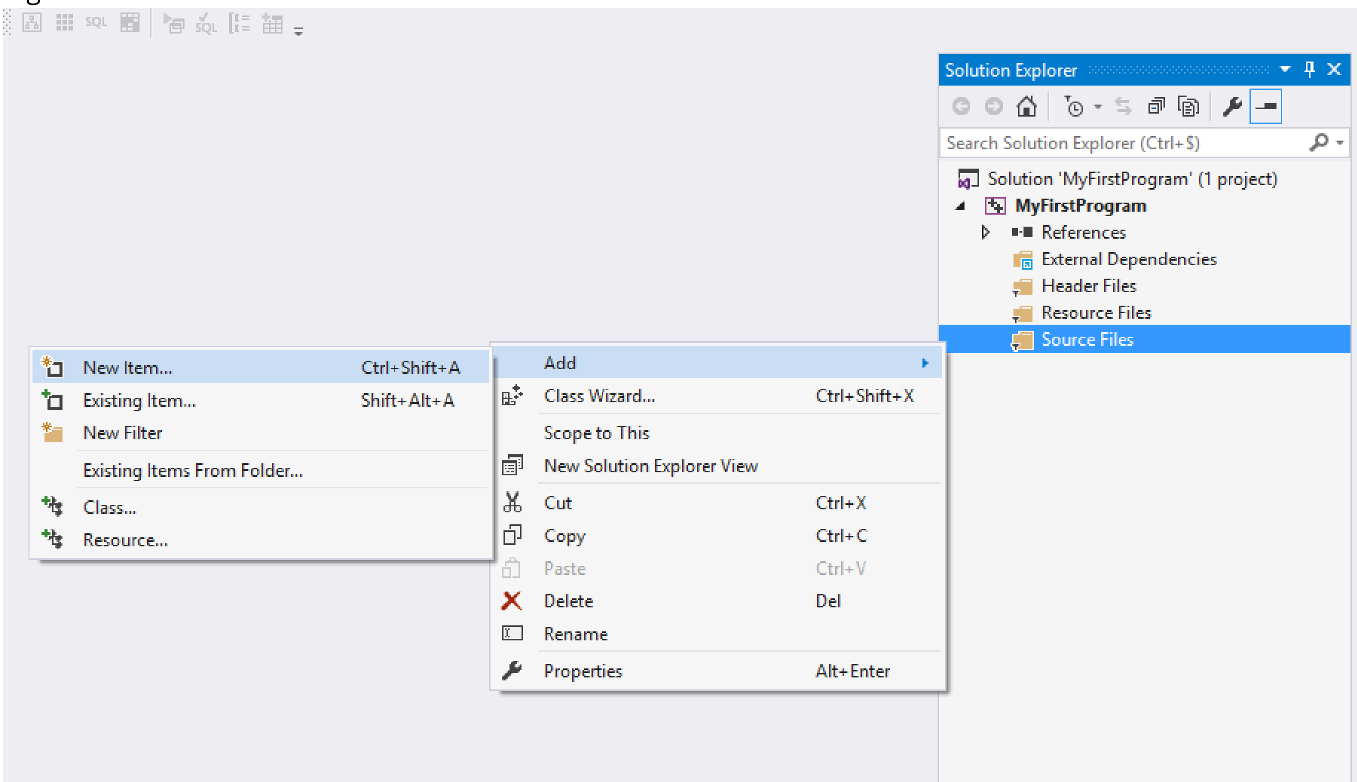
5. Click Ok
6. Click Next in the following window.



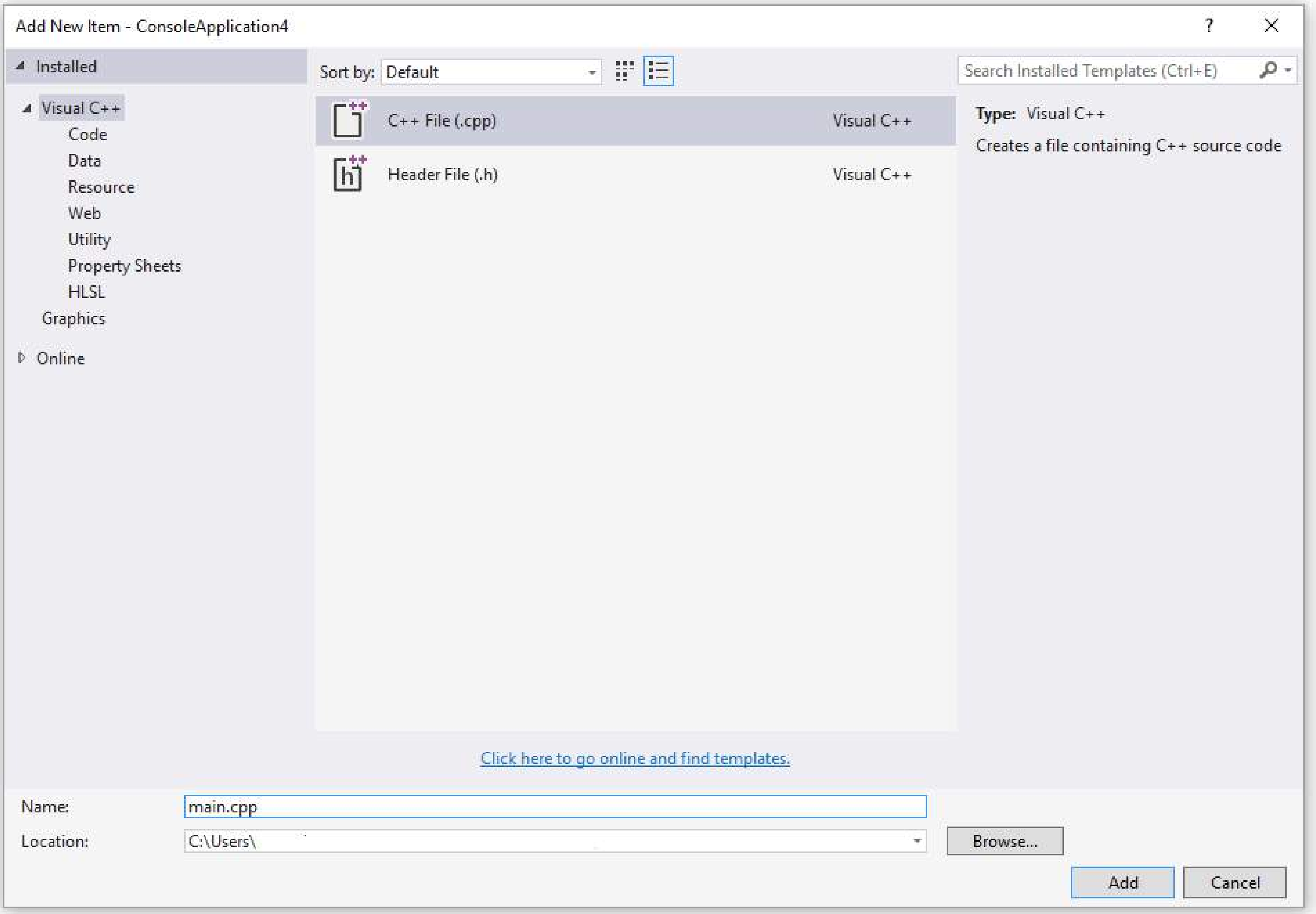
7. Check the `Empty project` box and then click Finish:



8. Right click on folder Source File then -> Add --> New Item :



9. Select C++ File and name the file main.cpp, then click Add:

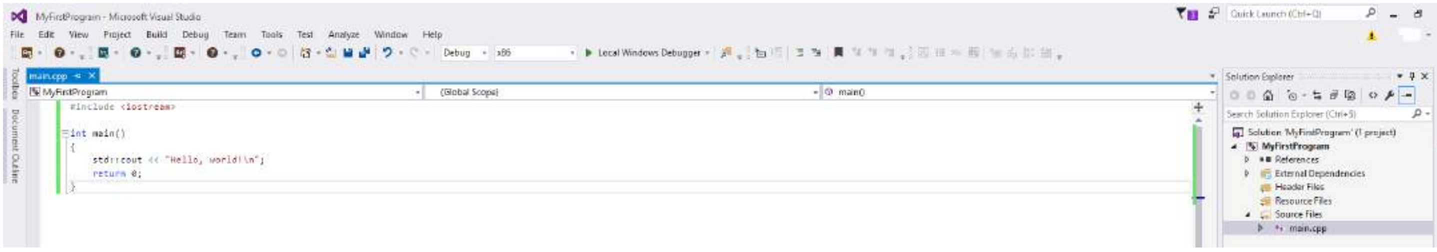


10: Copy and paste the following code in the new file main.cpp:

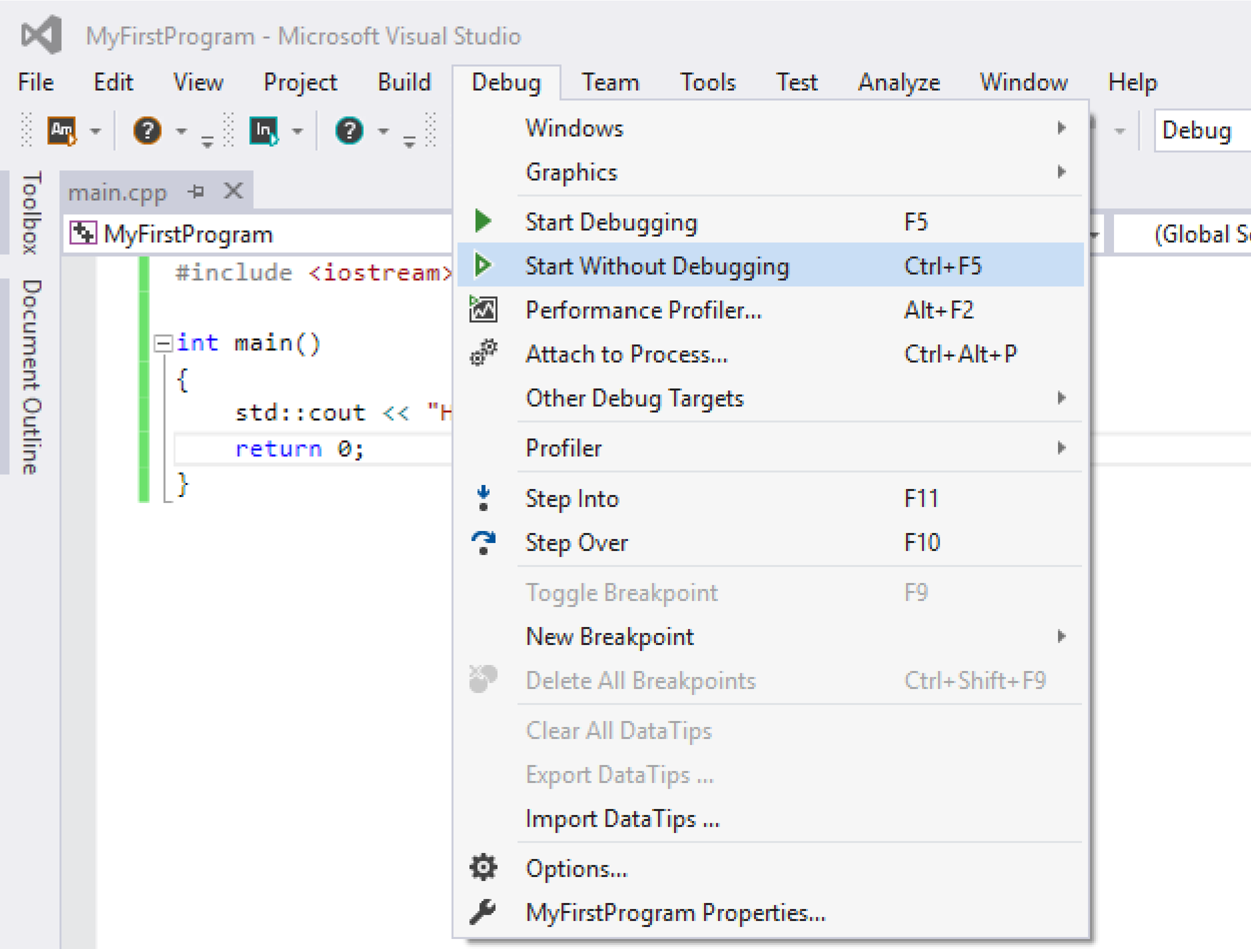
```
#include <iostream>

int main()
{
    std::cout << "Hello World!\n";
    return 0;
}
```

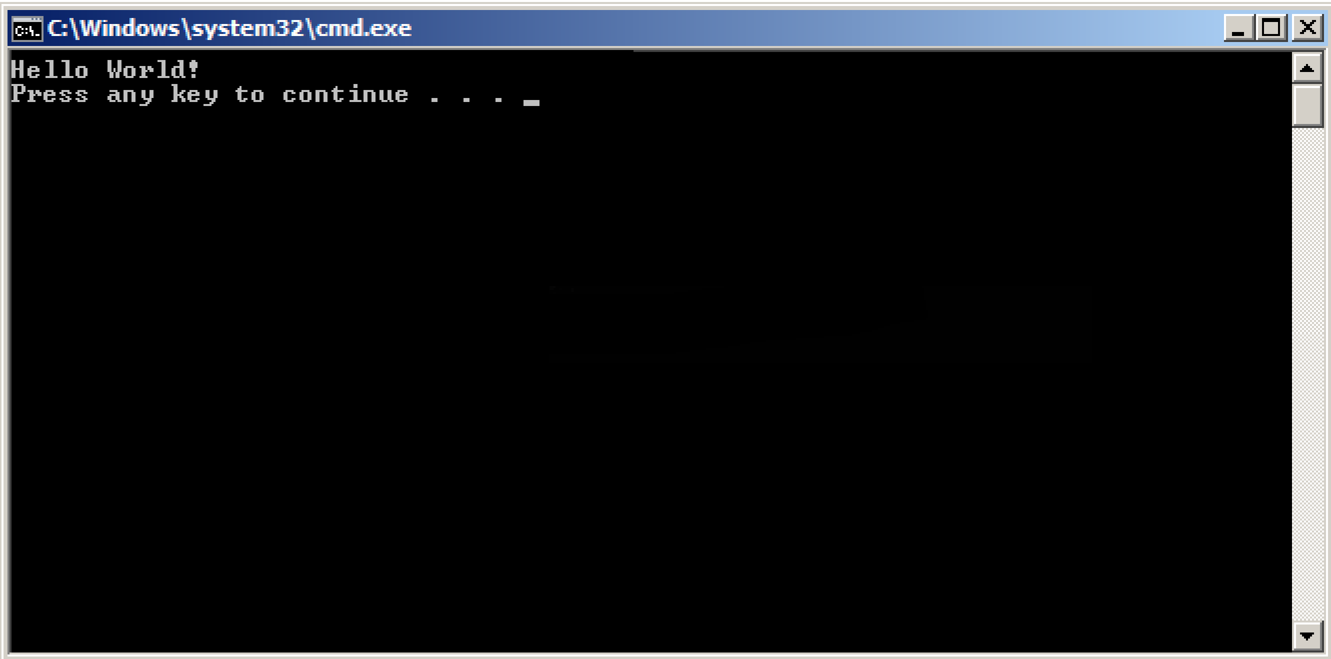
You environment should look like:



11. Click Debug -> Start **Without** Debugging (or press ctrl + F5) :



12. Done. You should get the following console output :



Section 138.3: Online Compilers

Various websites provide online access to C++ compilers. Online compiler's feature set vary significantly from site to site, but usually they allow to do the following:

- Paste your code into a web form in the browser.
- Select some compiler options and compile the code.
- Collect compiler and/or program output.

Online compiler website behavior is usually quite restrictive as they allow anyone to run compilers and execute arbitrary code on their server side, whereas ordinarily remote arbitrary code execution is considered as vulnerability.

Online compilers may be useful for the following purposes:

- Run a small code snippet from a machine which lacks C++ compiler (smartphones, tablets, etc.).
- Ensure that code compiles successfully with different compilers and runs the same way regardless the compiler it was compiled with.
- Learn or teach basics of C++.
- Learn modern C++ features (C++14 and C++17 in near future) when up-to-date C++ compiler is not available on local machine.
- Spot a bug in your compiler by comparison with a large set of other compilers. Check if a compiler bug was fixed in future versions, which are unavailable on your machine.
- Solve online judge problems.

What online compilers should **not** be used for:

- Develop full-featured (even small) applications using C++. Usually online compilers do not allow to link with third-party libraries or download build artifacts.
- Perform intensive computations. Sever-side computing resources are limited, so any user-provided program will be killed after a few seconds of execution. The permitted execution time is usually enough for testing and learning.
- Attack compiler server itself or any third-party hosts on the net.

Examples:

> Disclaimer: documentation author(s) are not affiliated with any resources listed below. Websites are listed alphabetically.

- http://codepad.org/ Online compiler with code sharing. Editing code after compiling with a source code warning or error does not work so well.
- http://coliru.stacked-crooked.com/ Online compiler for which you specify the command line. Provides both GCC and Clang compilers for use.
- http://cpp.sh/ - Online compiler with C++14 support. Does not allow you to edit compiler command line, but some options are available via GUI controls.
- https://gcc.godbolt.org/ - Provides a wide list of compiler versions, architectures, and disassembly output. Very useful when you need to inspect what your code compiles into by different compilers. GCC, Clang, MSVC (`CL`), Intel compiler (`icc`), ELLCC, and Zapcc are present, with one or more of these compilers available for the ARM, ARMv8 (as ARM64), Atmel AVR, MIPS, MIPS64, MSP430, PowerPC, x86, and x64 architecutres. Compiler command line arguments may be edited.
- https://ideone.com/ - Widely used on the Net to illustrate code snippet behavior. Provides both GCC and Clang for use, but doesn't allow you to edit the compiler command line.
- http://melpon.org/wandbox - Supports numerous Clang and GNU/GCC compiler versions.
- http://onlinegdb.com/ - An extremely minimalistic IDE that includes an editor, a compiler (gcc), and a debugger (gdb).
- http://rextester.com/ - Provides Clang, GCC, and Visual Studio compilers for both C and C++ (along with compilers for other languages), with the Boost library available for use.
- http://tutorialspoint.com/compile_cpp11_online.php - Full-featured UNIX shell with GCC, and a user-friendly project explorer.
- http://webcompiler.cloudapp.net/ - Online Visual Studio 2015 compiler, provided by Microsoft as part of

RiSE4fun.

Section 138.4: Compiling with Visual C++ (Command Line)

For programmers coming from GCC or Clang to Visual Studio, or programmers more comfortable with the command line in general, you can use the Visual C++ compiler from the command line as well as the IDE.

If you desire to compile your code from the command line in Visual Studio, you first need to set up the command line environment. This can be done either by opening the `Visual Studio Command Prompt`/`Developer Command Prompt`/`x86 Native Tools Command Prompt`/`x64 Native Tools Command Prompt` or similar (as provided by your version of Visual Studio), or at the command prompt, by navigating to the `VC` subdirectory of the compiler's install directory (typically `\Program Files (x86)\Microsoft Visual Studio x\VC`, where `x` is the version number (such as `10.0` for 2010, or `14.0` for 2015) and running the `VCVARSALL` batch file with a command-line parameter specified here.

Note that unlike GCC, Visual Studio doesn't provide a front-end for the linker (`link.exe`) via the compiler (`cl.exe`), but instead provides the linker as a separate program, which the compiler calls as it exits. `cl.exe` and `link.exe` can be used separately with different files and options, or `cl` can be told to pass files and options to `link` if both tasks are done together. Any linking options specified to `cl` will be translated into options for `link`, and any files not processed by `cl` will be passed directly to `link`. As this is mainly a simple guide to compiling with the Visual Studio command line, arguments for `link` will not be described at this time; if you need a list, see here.

Note that arguments to `cl` are case-sensitive, while arguments to `link` are not.

[Be advised that some of the following examples use the Windows shell "current directory" variable, `%cd%`, when specifying absolute path names. For anyone unfamiliar with this variable, it expands to the current working directory. From the command line, it will be the directory you were in when you ran `cl`, and is specified in the command prompt by default (if your command prompt is `C:\src>`, for example, then `%cd%` is `C:\src\`).]

Assuming a single source file named `main.cpp` in the current folder, the command to compile and link an unoptimised executable (useful for initial development and debugging) is (use either of the following):

```
cl main.cpp
// Generates object file "main.obj".
// Performs linking with "main.obj".
// Generates executable "main.exe".

cl /Od main.cpp
// Same as above.
// "/Od" is the "Optimisation: disabled" option, and is the default when no /O is specified.
```

Assuming an additional source file "niam.cpp" in the same directory, use the following:

```
cl main.cpp niam.cpp
// Generates object files "main.obj" and "niam.obj".
// Performs linking with "main.obj" and "niam.obj".
// Generates executable "main.exe".
```

You can also use wildcards, as one would expect:

```
cl main.cpp src\*.cpp
// Generates object file "main.obj", plus one object file for each ".cpp" file in folder
//  "%cd%\src".
// Performs linking with "main.obj", and every additional object file generated.
// All object files will be in the current folder.
```

```
// Generates executable "main.exe".
```

To rename or relocate the executable, use one of the following:

```
cl /o name main.cpp
// Generates executable named "name.exe".

cl /o folder\ main.cpp
// Generates executable named "main.exe", in folder "%cd%\folder".

cl /o folder\name main.cpp
// Generates executable named "name.exe", in folder "%cd%\folder".

cl /Fename main.cpp
// Same as "/o name".

cl /Fefolder\ main.cpp
// Same as "/o folder\".

cl /Fefolder\name main.cpp
// Same as "/o folder\name".
```

Both `/o` and `/Fe` pass their parameter (let's call it `o-param`) to `link` as `/OUT:o-param`, appending the appropriate extension (generally `.exe` or `.dll`) to "name" `o-param`s as necessary. While both `/o` and `/Fe` are to my knowledge identical in functionality, the latter is preferred for Visual Studio. `/o` is marked as deprecated, and appears to mainly be provided for programmers more familiar with GCC or Clang.

Note that while the space between `/o` and the specified folder and/or name is optional, there *cannot* be a space between `/Fe` and the specified folder and/or name.

Similarly, to produce an optimised executable (for use in production), use:

```
cl /O1 main.cpp
// Optimise for executable size.  Produces small programs, at the possible expense of slower
//  execution.

cl /O2 main.cpp
// Optimise for execution speed.  Produces fast programs, at the possible expense of larger
//  file size.

cl /GL main.cpp other.cpp
// Generates special object files used for whole-program optimisation, which allows CL to
//  take every module (translation unit) into consideration during optimisation.
// Passes the option "/LTCG" (Link-Time Code Generation) to LINK, telling it to call CL during
//  the linking phase to perform additional optimisations.  If linking is not performed at this
//  time, the generated object files should be linked with "/LTCG".
// Can be used with other CL optimisation options.
```

Finally, to produce a platform-specific optimized executable (for use in production on the machine with the specified architecture), choose the appropriate command prompt or `VCVARSALL` parameter for the target platform. `link` should detect the desired platform from the object files; if not, use the `/MACHINE` option to explicitly specify the target platform.

```
// If compiling for x64, and LINK doesn't automatically detect target platform:
cl main.cpp /link /machine:X64
```

Any of the above will produce an executable with the name specified by `/o` or `/Fe`, or if neither is provided, with a name identical to the first source or object file specified to the compiler.

```
cl a.cpp b.cpp c.cpp
// Generates "a.exe".

cl d.obj a.cpp q.cpp
// Generates "d.exe".

cl y.lib n.cpp o.obj
// Generates "n.exe".

cl /o yo zp.obj pz.cpp
// Generates "yo.exe".
```

To compile a file(s) without linking, use:

```
cl /c main.cpp
// Generates object file "main.obj".
```

This tells `cl` to exit without calling `link`, and produces an object file, which can later be linked with other files to produce a binary.

```
cl main.obj niam.cpp
// Generates object file "niam.obj".
// Performs linking with "main.obj" and "niam.obj".
// Generates executable "main.exe".

link main.obj niam.obj
// Performs linking with "main.obj" and "niam.obj".
// Generates executable "main.exe".
```

There are other valuable command line parameters as well, which it would be very useful for users to know:

```
cl /EHsc main.cpp
// "/EHsc" specifies that only standard C++ ("synchronous") exceptions will be caught,
//  and `extern "C"` functions will not throw exceptions.
// This is recommended when writing portable, platform-independent code.

cl /clr main.cpp
// "/clr" specifies that the code should be compiled to use the common language runtime,
//  the .NET Framework's virtual machine.
// Enables the use of Microsoft's C++/CLI language in addition to standard ("native") C++,
//  and creates an executable that requires .NET to run.

cl /Za main.cpp
// "/Za" specifies that Microsoft extensions should be disabled, and code should be
//  compiled strictly according to ISO C++ specifications.
// This is recommended for guaranteeing portability.

cl /Zi main.cpp
// "/Zi" generates a program database (PDB) file for use when debugging a program, without
//  affecting optimisation specifications, and passes the option "/DEBUG" to LINK.

cl /LD dll.cpp
// "/LD" tells CL to configure LINK to generate a DLL instead of an executable.
// LINK will output a DLL, in addition to an LIB and EXP file for use when linking.
// To use the DLL in other programs, pass its associated LIB to CL or LINK when compiling those
//  programs.

cl main.cpp /link /LINKER_OPTION
// "/link" passes everything following it directly to LINK, without parsing it in any way.
```

```
// Replace "/LINKER_OPTION" with any desired LINK option(s).
```

For anyone more familiar with *nix systems and/or GCC/Clang, `cl`, `link`, and other Visual Studio command line tools can accept parameters specified with a hyphen (such as `-c`) instead of a slash (such as `/c`). Additionally, Windows recognises either a slash or a backslash as a valid path separator, so *nix-style paths can be used as well. This makes it easy to convert simple compiler command lines from `g++` or `clang++` to `cl`, or vice versa, with minimal changes.

```
g++ -o app src/main.cpp
cl  -o app src/main.cpp
```

Of course, when porting command lines that use more complex `g++` or `clang++` options, you need to look up equivalent commands in the applicable compiler documentations and/or on resource sites, but this makes it easier to get things started with minimal time spent learning about new compilers.

In case you need specific language features for your code, a specific release of MSVC was required. From Visual C++ 2015 Update 3 on it is possible to choose the version of the standard to compile with via the `/std` flag. Possible values are `/std:c++14` and `/std:c++latest` (`/std:c++17` will follow soon).

Note: In older versions of this compiler, specific feature flags were available however this was mostly used for previews of new features.

Section 138.5: Compiling with Clang

As the Clang front-end is designed for being compatible with GCC, most programs that can be compiled via GCC will compile when you swap `g++` by `clang++` in the build scripts. If no `-std=version` is given, gnu11 will be used.

Windows users who are used to MSVC can swap `cl.exe` with `clang-cl.exe`. By default, clang tries to be compatible with the highest version of MSVC that has been installed.

In the case of compiling with visual studio, clang-cl can be used by changing the `Platform toolset` in the project properties.

In both cases, clang is only compatible via its front-end, though it also tries to generate binary compatible object files. Users of clang-cl should note that the compatibility with MSVC is not complete yet.

To use clang or clang-cl, one could use the default installation on certain Linux distributions or those bundled with IDEs (like XCode on Mac). For other versions of this compiler or on platforms which don't have this installed, this can be download from the official download page.

If you're using CMake to build your code you can usually switch the compiler by setting the `CC` and `CXX` environment variables like this:

```
mkdir build
cd build
CC=clang CXX=clang++ cmake ..
cmake --build .
```

See also introduction to Cmake.

Section 138.6: The C++ compilation process

When you develop a C++ program, the next step is to compile the program before running it. The compilation is the process which converts the program written in human readable language like C, C++ etc into a machine code,

directly understood by the Central Processing Unit. For example, if you have a C++ source code file named prog.cpp and you execute the compile command,

```
g++ -Wall -ansi -o prog prog.cpp
```

There are 4 main stages involved in creating an executable file from the source file.

1. The C++ the preprocessor takes a C++ source code file and deals with the headers(#include), macros(#define) and other preprocessor directives.
2. The expanded C++ source code file produced by the C++ preprocessor is compiled into the assembly language for the platform.
3. The assembler code generated by the compiler is assembled into the object code for the platform.
4. The object code file produced by the assembler is linked together
with the object code files for any library functions used to produce either a library or an executable file.

Preprocessing

The preprocessor handles the preprocessor directives, like #include and #define. It is agnostic of the syntax of C++, which is why it must be used with care.

It works on one C++ source file at a time by replacing #include directives with the content of the respective files (which is usually just declarations), doing replacement of macros (#define), and selecting different portions of text depending of #if, #ifdef and #ifndef directives.

The preprocessor works on a stream of preprocessing tokens. Macro substitution is defined as replacing tokens with other tokens (the operator ## enables merging two tokens when it make sense).

After all this, the preprocessor produces a single output that is a stream of tokens resulting from the transformations described above. It also adds some special markers that tell the compiler where each line came from so that it can use those to produce sensible error messages.

Some errors can be produced at this stage with clever use of the #if and #error directives.

By using below compiler flag, we can stop the process at preprocessing stage.

```
g++ -E prog.cpp
```

Compilation

The compilation step is performed on each output of the preprocessor. The compiler parses the pure C++ source code (now without any preprocessor directives) and converts it into assembly code. Then invokes underlying back-end(assembler in toolchain) that assembles that code into machine code producing actual binary file in some format(ELF, COFF, a.out, ...). This object file contains the compiled code (in binary form) of the symbols defined in the input. Symbols in object files are referred to by name.

Object files can refer to symbols that are not defined. This is the case when you use a declaration, and don't provide a definition for it. The compiler doesn't mind this, and will happily produce the object file as long as the source code is well-formed.

Compilers usually let you stop compilation at this point. This is very useful because with it you can compile each source code file separately. The advantage this provides is that you don't need to recompile everything if you only change a single file.

The produced object files can be put in special archives called static libraries, for easier reusing later on.

It's at this stage that "regular" compiler errors, like syntax errors or failed overload resolution errors, are reported.

In order to stop the process after the compile step, we can use the -S option:

```
g++ -Wall -ansi -S prog.cpp
```

Assembling

The assembler creates object code. On a UNIX system you may see files with a .o suffix (.OBJ on MSDOS) to indicate object code files. In this phase the assembler converts those object files from assembly code into machine level instructions and the file created is a relocatable object code. Hence, the compilation phase generates the relocatable object program and this program can be used in different places without having to compile again.

To stop the process after the assembly step, you can use the -c option:

```
g++ -Wall -ansi -c prog.cpp
```

Linking

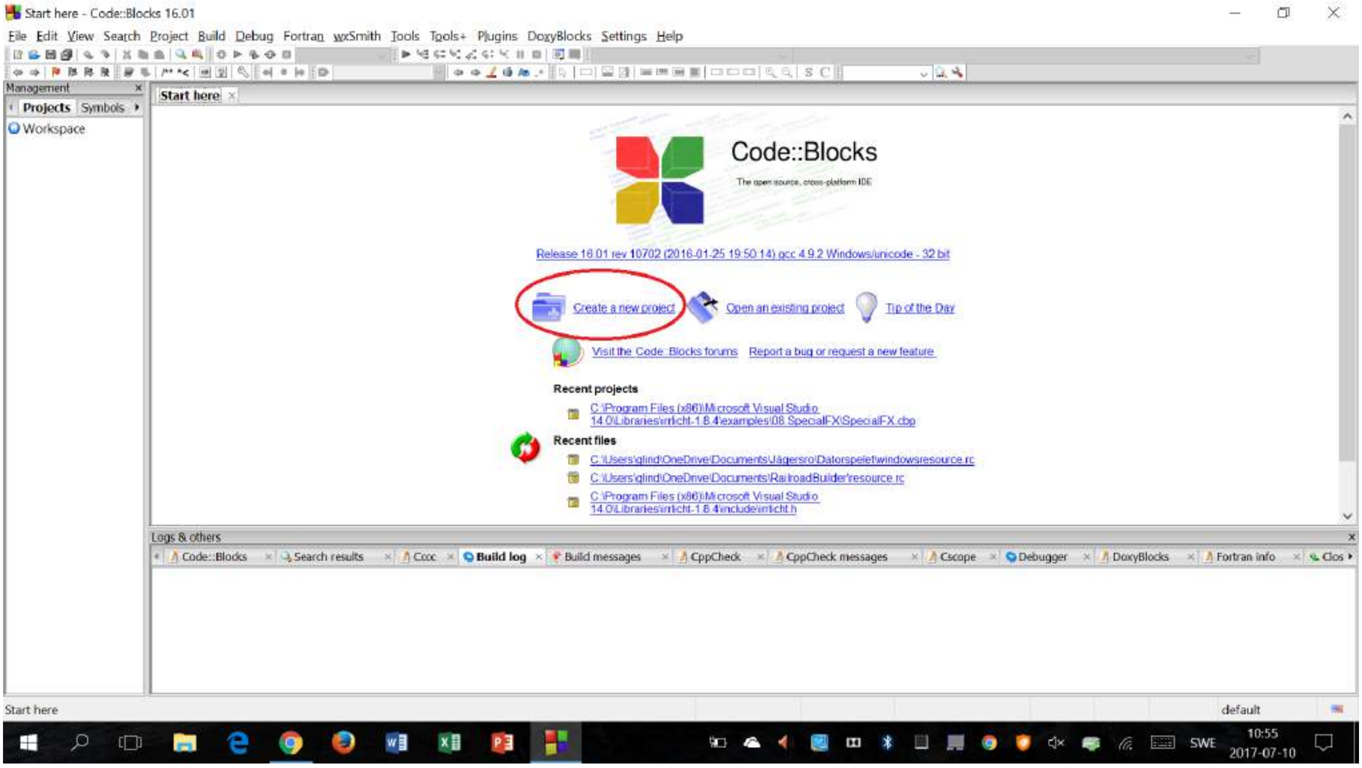
The linker is what produces the final compilation output from the object files the assembler produced. This output can be either a shared (or dynamic) library (and while the name is similar, they don't have much in common with static libraries mentioned earlier) or an executable.

It links all the object files by replacing the references to undefined symbols with the correct addresses. Each of these symbols can be defined in other object files or in libraries. If they are defined in libraries other than the standard library, you need to tell the linker about them.

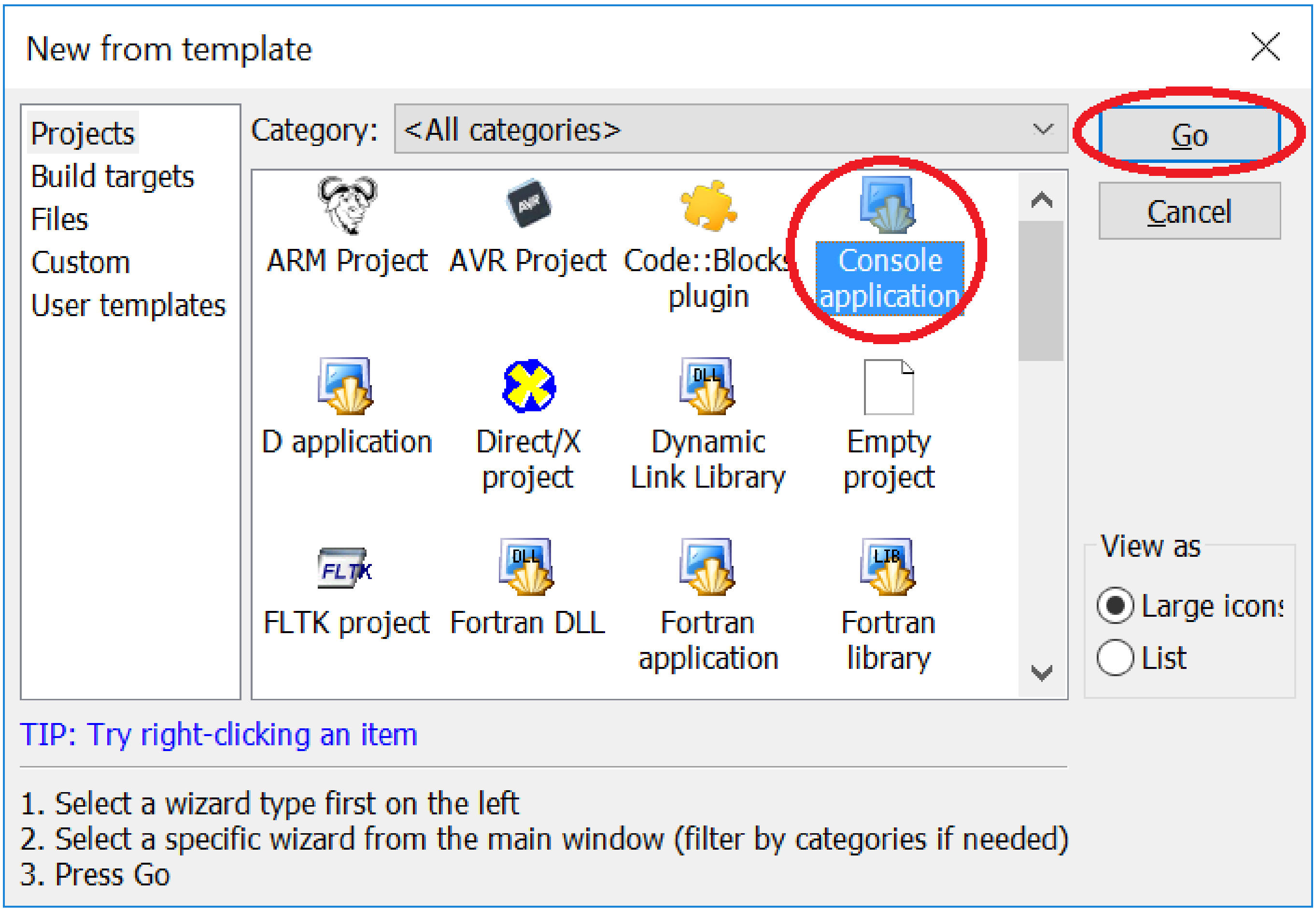
At this stage the most common errors are missing definitions or duplicate definitions. The former means that either the definitions don't exist (i.e. they are not written), or that the object files or libraries where they reside were not given to the linker. The latter is obvious: the same symbol was defined in two different object files or libraries.

Section 138.7: Compiling with Code::Blocks (Graphical interface)

1. Download and install Code::Blocks here. If you're on Windows, be careful to select a file for which the name contains `mingw`, the other files don't install any compiler.
2. Open Code::Blocks and click on "Create a new project":



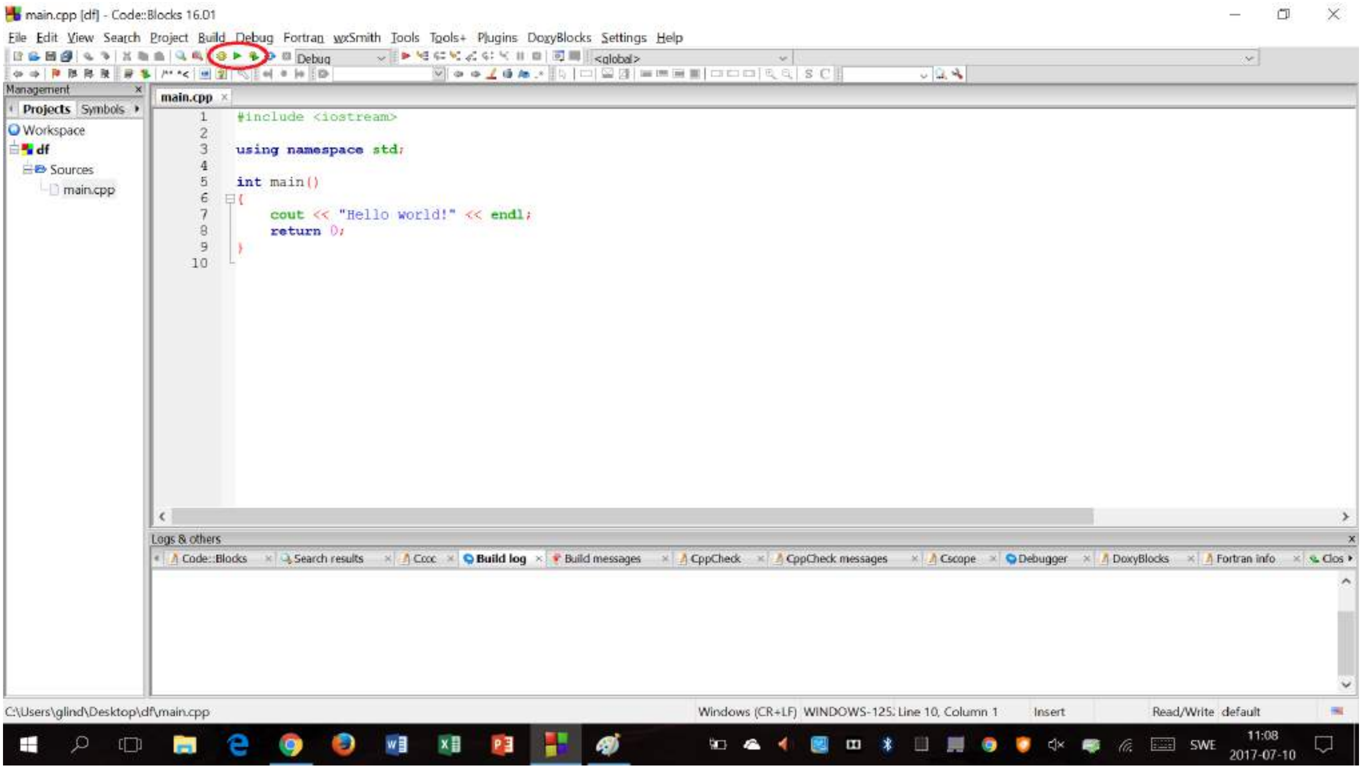
3. Select "Console application" and click "Go":



4. Click "Next", select "C++", click "Next", select a name for your project and choose a folder to save it in, click "Next" and then click "Finish".

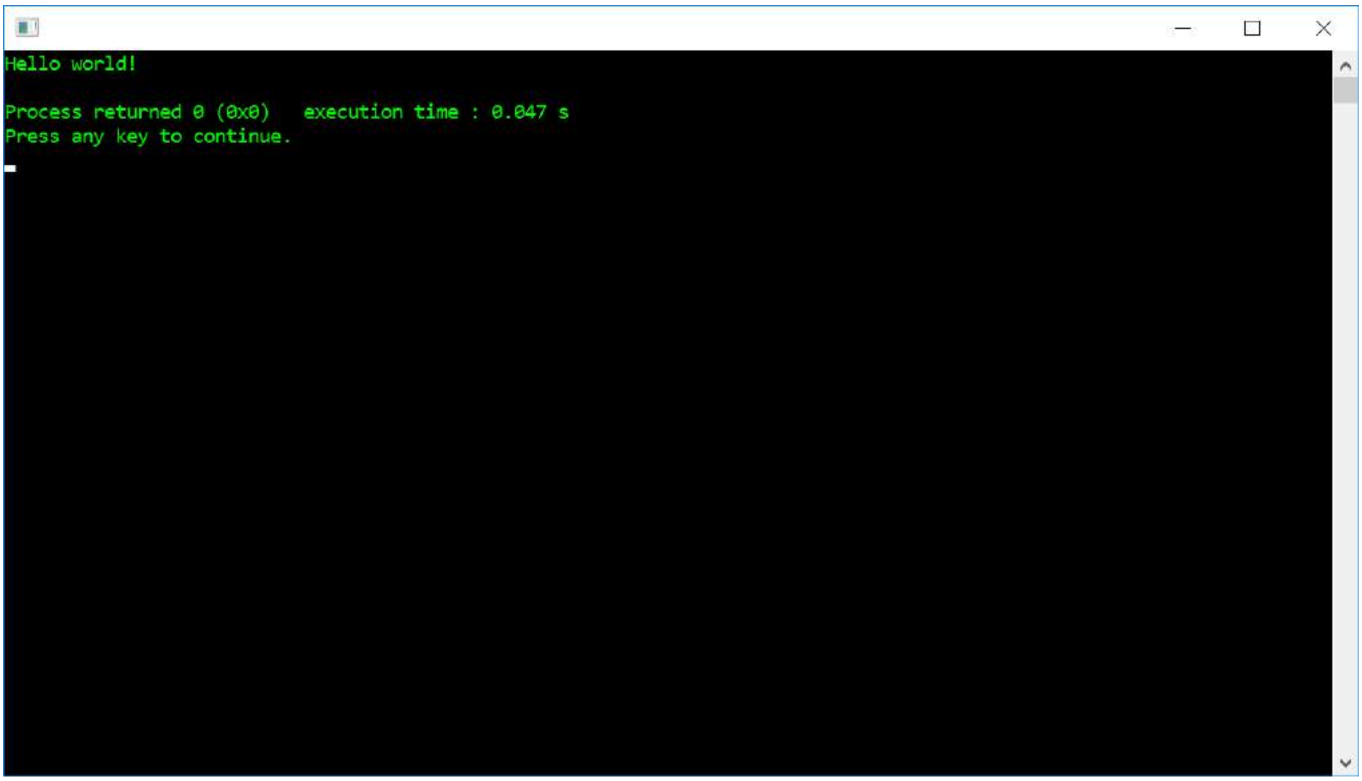
5. Now you can edit and compile your code. A default code that prints "Hello world!" in the console is already

there. To compile and/or run your program, press one of the three compile/run buttons in the toolbar:



To compile without running, press ⚙, to run without compiling again, press ▶ and to compile and then run, press ⚙▶.

Compiling and running the default "Hello world!" code gives the following result:



Chapter 139: Common compile/linker errors (GCC)

Section 139.1: undefined reference to `***'

This linker error happens, if the linker can't find a used symbol. Most of the time, this happens if a used library is not linked against.

qmake:

```
LIBS += nameOfLib
```

cmake:

```
TARGET_LINK_LIBRARIES(target nameOfLib)
```

g++ call:

```
g++ -o main main.cpp -Llibrary/dir -lnameOfLib
```

One might also forget to compile and link all used `.cpp` files (functionsModule.cpp defines the needed function):

```
g++  -o binName main.o functionsModule.o
```

Section 139.2: error: '***' was not declared in this scope

This error happens if a unknown object is used.

Variables

Not compiling:

```
#include <iostream>

int main(int argc, char *argv[])
{
    {
        int i = 2;
    }

    std::cout << i << std::endl; // i is not in the scope of the main function

    return 0;
}
```

Fix:

```
#include <iostream>

int main(int argc, char *argv[])
{
    {
        int i = 2;
        std::cout << i << std::endl;
```

```
    }

    return 0;
}
```

Functions

> Most of the time this error occurs if the needed header is not included (e.g. using `std::cout` without `#include <iostream>`)

Not compiling:

```
#include <iostream>

int main(int argc, char *argv[])
{
    doCompile();

    return 0;
}

void doCompile()
{
    std::cout << "No!" << std::endl;
}
```

Fix:

```
#include <iostream>

void doCompile(); // forward declare the function

int main(int argc, char *argv[])
{
    doCompile();

    return 0;
}

void doCompile()
{
    std::cout << "No!" << std::endl;
}
```

Or:

```
#include <iostream>

void doCompile() // define the function before using it
{
    std::cout << "No!" << std::endl;
}

int main(int argc, char *argv[])
{
    doCompile();

    return 0;
```

```
}
```

Note: The compiler interprets the code from top to bottom (simplification). Everything must be at least declared (or defined) before usage.

Section 139.3: fatal error: ***: No such file or directory

The compiler can't find a file (a source file uses `#include "someFile.hpp"`).

qmake:

```
INCLUDEPATH += dir/Of/File
```

cmake:

```
include_directories(dir/Of/File)
```

g++ call:

```
g++ -o main main.cpp -Idir/Of/File
```

Chapter 140: More undefined behaviors in C++

More examples on how C++ can go wrong.

Continuation from Undefined Behavior

Section 140.1: Referring to non-static members in initializer lists

Referring to non-static members in initializer lists before the constructor has started executing can result in undefined behavior. This results since not all members are constructed at this time. From the standard draft:

> §12.7.1: For an object with a non-trivial constructor, referring to any non-static member or base class of the object before the constructor begins execution results in undefined behavior.

Example

```
struct W { int j; };
struct X : public virtual W { };
struct Y {
   int *p;
   X x;
   Y() : p(&x.j) { // undefined, x is not yet constructed
   }
};
```

Chapter 141: Unit Testing in C++

Unit testing is a level in software testing that validates the behavior and correctness of units of code.

In C++, "units of code" often refer to either classes, functions, or groups of either. Unit testing is often performed using specialized "testing frameworks" or "testing libraries" that often use non-trivial syntax or usage patterns.

This topic will review different strategies and unit testing libraries or frameworks.

Section 141.1: Google Test

Google Test is a C++ testing framework maintained by Google. It requires building the `gtest` library and linking it to your testing framework when building a test case file.

Minimal Example

```
// main.cpp

#include <gtest/gtest.h>
#include <iostream>

// Google Test test cases are created using a C++ preprocessor macro
// Here, a "test suite" name and a specific "test name" are provided.
TEST(module_name, test_name) {
    std::cout << "Hello world!" << std::endl;
    // Google Test will also provide macros for assertions.
    ASSERT_EQ(1+1, 2);
}

// Google Test can be run manually from the main() function
// or, it can be linked to the gtest_main library for an already
// set-up main() function primed to accept Google Test test cases.
int main(int argc, char** argv) {
    ::testing::InitGoogleTest(&argc, argv);

    return RUN_ALL_TESTS();
}

// Build command: g++ main.cpp -lgtest
```

Section 141.2: Catch

Catch is a header only library that allows you to use both TDD and BDD unit test style.

The following snippet is from the Catch documentation page at this link:

```
SCENARIO( "vectors can be sized and resized", "[vector]" ) {
    GIVEN( "A vector with some items" ) {
        std::vector v( 5 );

        REQUIRE( v.size() == 5 );
        REQUIRE( v.capacity() >= 5 );

        WHEN( "the size is increased" ) {
            v.resize( 10 );

            THEN( "the size and capacity change" ) {
                REQUIRE( v.size() == 10 );
```

```
                REQUIRE( v.capacity() >= 10 );
            }
        }
        WHEN( "the size is reduced" ) {
            v.resize( 0 );

            THEN( "the size changes but not capacity" ) {
                REQUIRE( v.size() == 0 );
                REQUIRE( v.capacity() >= 5 );
            }
        }
        WHEN( "more capacity is reserved" ) {
            v.reserve( 10 );

            THEN( "the capacity changes but not the size" ) {
                REQUIRE( v.size() == 5 );
                REQUIRE( v.capacity() >= 10 );
            }
        }
        WHEN( "less capacity is reserved" ) {
            v.reserve( 0 );

            THEN( "neither size nor capacity are changed" ) {
                REQUIRE( v.size() == 5 );
                REQUIRE( v.capacity() >= 5 );
            }
        }
    }
}
```

Conveniently, these tests will be reported as follows when run:

Scenario: vectors can be sized and resized Given: A vector with some items When: more capacity is reserved Then: the capacity changes but not the size

Chapter 142: C++ Debugging and Debug-prevention Tools & Techniques

A lot of time from C++ developers is spent debugging. This topic is meant to assist with this task and give inspiration for techniques. Don't expect an extensive list of issues and solutions fixed by the tools or a manual on the mentioned tools.

Section 142.1: Static analysis

Static analysis is the technique in which on checks the code for patterns linked to known bugs. Using this technique is less time consuming than a code review, though, its checks are only limited to those programmed in the tool.

Checks can include the incorrect semi-colon behind the if-statement (`if (var);`) till advanced graph algorithms which determine if a variable is not initialized.

Compiler warnings

Enabling static analysis is easy, the most simplistic version is already build-in in your compiler:

- `clang++ -Wall -Weverything -Werror ...`
- `g++ -Wall -Weverything -Werror ...`
- `cl.exe /W4 /WX ...`

If you enable these options, you will notice that each compiler will find bugs the others don't and that you will get errors on techniques which might be valid or valid in a specific context. `while (staticAtomicBool);` might be acceptable even if `while (localBool);` ain't.

So unlike code review, you are fighting a tool which understands your code, tells you a lot of useful bugs and sometimes disagrees with you. In this last case, you might have to suppress the warning locally.

As the options above enable all warnings, they might enable warnings you don't want. (Why should your code be C++98 compatible?) If so, you can simply disable that specific warning:

- `clang++ -Wall -Weverything -Werror -Wno-errortoaccept ...`
- `g++ -Wall -Weverything -Werror -Wno-errortoaccept ...`
- `cl.exe /W4 /WX /wd<no of warning>...`

Where compiler warnings assist you during development, they slow down compilation quite a bit. That is why you might not always want to enable them by default. Either you run them by default or you enable some continuous integration with the more expensive checks (or all of them).

External tools

If you decide to have some continuous integration, the use of other tools ain't such a stretch. A tool like clang-tidy has an list of checks which covers a wide range of issues, some examples:

- Actual bugs
  - Prevention of slicing
  - Asserts with side effects
- Readability checks
  - Misleading indentation
  - Check identifier naming
- Modernization checks

  - Use make_unique()
  - Use nullptr
- Performance checks
  - Find unneeded copies
  - Find inefficient algorithm calls

The list might not be that large, as Clang already has a lot of compiler warnings, however it will bring you one step closer to a high quality code base.

Other tools

Other tools with similar purpose exist, like:

- the visual studio static analyzer as external tool
- clazy, a Clang compiler plugin for checking Qt code

Conclusion

A lot static analysis tools exist for C++, both build-in in the compiler as external tools. Trying them out doesn't take that much time for easy setups and they will find bugs you might miss in code review.

Section 142.2: Segfault analysis with GDB

Lets use the same code as above for this example.

```
#include <iostream>

void fail() {
    int *p1;
    int *p2(NULL);
    int *p3 = p1;
    if (p3) {
        std::cout << *p2 << std::endl;
    }
}

int main() {
    fail();
}
```

First lets compile it

```
g++ -g -o main main.cpp
```

Lets run it with gdb

```
gdb ./main
```

Now we will be in gdb shell. Type run.

```
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/opencog/code-snippets/stackoverflow/a.out

Program received signal SIGSEGV, Segmentation fault.
0x0000000000400850 in fail () at debugging_with_gdb.cc:11
```

```
11              std::cout << *p2 << std::endl;
```

We see the segmentation fault is happening at line 11. So the only variable being used at this line is pointer p2. Lets examine its content typing print.

```
(gdb) print p2
$1 = (int *) 0x0
```

Now we see that p2 was initialized to 0x0 which stands for NULL. At this line, we know that we are trying to dereference a NULL pointer. So we go and fix it.

Section 142.3: Clean code

Debugging starts with understanding the code you are trying to debug.

Bad code:

```
int main() {
    int value;
    std::vector<int> vectorToSort;
    vectorToSort.push_back(42); vectorToSort.push_back(13);
    for (int i = 52; i; i = i - 1)
    {
    vectorToSort.push_back(i *2);
    }
    /// Optimized for sorting small vectors
    if (vectorToSort.size() == 1);
    else
        {
        if (vectorToSort.size() <= 2)
            std::sort(vectorToSort.begin(), std::end(vectorToSort));
        }
    for (value : vectorToSort) std::cout << value << ' ';
return 0; }
```

Better code:

```
std::vector<int> createSemiRandomData() {
    std::vector<int> data;
    data.push_back(42);
    data.push_back(13);
    for (int i = 52; i; --i)
        vectorToSort.push_back(i *2);
    return data;
}

/// Optimized for sorting small vectors
void sortVector(std::vector &v) {
    if (vectorToSort.size() == 1)
        return;
    if (vectorToSort.size() > 2)
        return;

    std::sort(vectorToSort.begin(), vectorToSort.end());
}

void printVector(const std::vector<int> &v) {
    for (auto i : v)
        std::cout << i << ' ';
```

```
}

int main() {
    auto vectorToSort = createSemiRandomData();
    sortVector(std::ref(vectorToSort));
    printVector(vectorToSort);

    return 0;
 }
```

Regardless of the coding styles you prefer and use, having a consistent coding (and formatting) style will help you understanding the code.

Looking at the code above, one can identify a couple of improvements to improve readability and debuggability:

The use of separate functions for separate actions

The use of separate functions allow you to skip over some functions in the debugger if you ain't interested in the details. In this specific case, you might not be interested in the creation or printing of the data and only want to step into the sorting.

Another advantage is that you need to read less code (and memorize it) while stepping through the code. You now only need to read 3 lines of code in `main()` in order to understand it, instead of the whole function.

The third advantage is that you simply have less code to look at, which helps a trained eye in spotting this bug within seconds.

Using consistent formatting/constructions

The use of consistent formatting and constructions will remove clutter from the code making it easier to focus on the code instead of text. A lot of discussions have been fed on the 'right' formatting style. Regardless of that style, having a single consistent style in the code will improve familiarity and make it easier to focus on the code.

As formatting code is time consuming task, it is recommended to use a dedicated tool for this. Most IDEs have at least some kind of support for this and can do formatting more consistent than humans.

You might note that the style is not limited to spaces and newlines as we no longer mix the free-style and the member functions to get begin/end of the container. (`v.begin()` vs `std::end(v)`).

Point attention to the important parts of your code.

Regardless of the style you determine to choose, the above code contains a couple of markers which might give you a hint on what might be important:

- A comment stating `optimized`, this indicates some fancy techniques
- Some early returns in `sortVector()` indicate that we are doing something special
- The `std::ref()` indicates that something is going on with the `sortVector()`

Conclusion

Having clean code will help you understanding the code and will reduce the time you need to debug it. In the second example, a code reviewer might even spot the bug at first glance, while the bug might be hidden in the details in the first one. (PS: The bug is in the compare with `2`.)

Chapter 143: Optimization in C++

Section 143.1: Introduction to performance

C and C++ are well known as high-performance languages - largely due to the heavy amount of code customization, allowing a user to specify performance by choice of structure.

When optimizing it is important to benchmark relevant code and completely understand how the code will be used.

Common optimization mistakes include:

- **Premature optimization:** Complex code may perform *worse* after optimization, wasting time and effort. First priority should be to write *correct* and *maintainable* code, rather than optimized code.
- **Optimization for the wrong use case:** Adding overhead for the 1% might not be worth the slowdown for the other 99%
- **Micro-optimization:** Compilers do this very efficiently and micro-optimization can even hurt the compilers ability to further optimize the code

Typical optimization goals are:

- To do less work
- To use more efficient algorithms/structures
- To make better use of hardware

Optimized code can have negative side effects, including:

- Higher memory usage
- Complex code -being difficult to read or maintain
- Compromised API and code design

Section 143.2: Empty Base Class Optimization

An object cannot occupy less than 1 byte, as then the members of an array of this type would have the same address. Thus `sizeof(T)>=1` always holds. It's also true that a derived class cannot be smaller than *any of* its base classes. However, when the base class is empty, its size is not necessarily added to the derived class:

```
class Base {};

class Derived : public Base
{
public:
    int i;
};
```

In this case, it's not required to allocate a byte for `Base` within `Derived` to have a distinct address per type per object. If empty base class optimization is performed (and no padding is required), then `sizeof(Derived) == sizeof(int)`, that is, no additional allocation is done for the empty base. This is possible with multiple base classes as well (in C++, multiple bases cannot have the same type, so no issues arise from that).

Note that this can only be performed if the first member of `Derived` differs in type from any of the base classes. This includes any direct or indirect common bases. If it's the same type as one of the bases (or there's a common base), at least allocating a single byte is required to ensure that no two distinct objects of the same type have the same address.

Section 143.3: Optimizing by executing less code

The most straightforward approach to optimizing is by executing less code. This approach usually gives a fixed speed-up without changing the time complexity of the code.

Even though this approach gives you a clear speedup, this will only give noticable improvements when the code is called a lot.

Removing useless code

```
void func(const A *a); // Some random function

// useless memory allocation + deallocation for the instance
auto a1 = std::make_unique<A>();
func(a1.get());

// making use of a stack object prevents
auto a2 = A{};
func(&a2);
```

Version ≥ C++14

From C++14, compilers are allowed to optimize this code to remove the allocation and matching deallocation.

Doing code only once

```
std::map<std::string, std::unique_ptr<A>> lookup;
// Slow insertion/lookup
// Within this function, we will traverse twice through the map lookup an element
// and even a thirth time when it wasn't in
const A *lazyLookupSlow(const std::string &key) {
    if (lookup.find(key) != lookup.cend())
        lookup.emplace_back(key, std::make_unique<A>());
    return lookup[key].get();
}

// Within this function, we will have the same noticeable effect as the slow variant while going at
double speed as we only traverse once through the code
const A *lazyLookupSlow(const std::string &key) {
    auto &value = lookup[key];
    if (!value)
        value = std::make_unique<A>();
    return value.get();
}
```

A similar approach to this optimization can be used to implement a stable version of `unique`

```
std::vector<std::string> stableUnique(const std::vector<std::string> &v) {
    std::vector<std::string> result;
    std::set<std::string> checkUnique;
    for (const auto &s : v) {
        // As insert returns if the insertion was successful, we can deduce if the element was
already in or not
        // This prevents an insertion, which will traverse through the map for every unique element
        // As a result we can almost gain 50% if v would not contain any duplicates
        if (checkUnique.insert(s).second)
            result.push_back(s);
    }
    return result;
}
```

Preventing useless reallocating and copying/moving

In the previous example, we already prevented lookups in the std::set, however the `std::vector` still contains a growing algorithm, in which it will have to realloc its storage. This can be prevented by first reserving for the right size.

```
std::vector<std::string> stableUnique(const std::vector<std::string> &v) {
    std::vector<std::string> result;
    // By reserving 'result', we can ensure that no copying or moving will be done in the vector
    // as it will have capacity for the maximum number of elements we will be inserting
    // If we make the assumption that no allocation occurs for size zero
    // and allocating a large block of memory takes the same time as a small block of memory
    // this will never slow down the program
    // Side note: Compilers can even predict this and remove the checks the growing from the
generated code
    result.reserve(v.size());
    std::set<std::string> checkUnique;
    for (const auto &s : v) {
        // See example above
        if (checkUnique.insert(s).second)
            result.push_back(s);
    }
    return result;
}
```

Section 143.4: Using efficient containers

Optimizing by using the right data structures at the right time can change the time-complexity of the code.

```
// This variant of stableUnique contains a complexity of N log(N)
// N > number of elements in v
// log(N) > insert complexity of std::set
std::vector<std::string> stableUnique(const std::vector<std::string> &v) {
    std::vector<std::string> result;
    std::set<std::string> checkUnique;
    for (const auto &s : v) {
        // See Optimizing by executing less code
        if (checkUnique.insert(s).second)
            result.push_back(s);
    }
    return result;
}
```

By using a container which uses a different implementation for storing its elements (hash container instead of tree), we can transform our implementation to complexity N. As a side effect, we will call the comparison operator for std::string less, as it only has to be called when the inserted string should end up in the same bucket.

```
// This variant of stableUnique contains a complexity of N
// N > number of elements in v
// 1 > insert complexity of std::unordered_set
std::vector<std::string> stableUnique(const std::vector<std::string> &v) {
    std::vector<std::string> result;
    std::unordered_set<std::string> checkUnique;
    for (const auto &s : v) {
        // See Optimizing by executing less code
        if (checkUnique.insert(s).second)
            result.push_back(s);
    }
    return result;
```

```
}
```

Section 143.5: Small Object Optimization

Small object optimization is a technique which is used within low level data structures, for instance the `std::string` (Sometimes referred to as Short/Small String Optimization). It's meant to use stack space as a buffer instead of some allocated memory in case the content is small enough to fit within the reserved space.

By adding extra memory overhead and extra calculations, it tries to prevent an expensive heap allocation. The benefits of this technique are dependent on the usage and can even hurt performance if incorrectly used.

Example

A very naive way of implementing a string with this optimization would the following:

```
#include <cstring>

class string final
{
    constexpr static auto SMALL_BUFFER_SIZE = 16;

    bool _isAllocated{false};                       ///< Remember if we allocated memory
    char *_buffer{nullptr};                         ///< Pointer to the buffer we are using
    char _smallBuffer[SMALL_BUFFER_SIZE]= {'\0'};   ///< Stack space used for SMALL OBJECT
OPTIMIZATION

public:
    ~string()
    {
        if (_isAllocated)
            delete [] _buffer;
    }

    explicit string(const char *cStyleString)
    {
        auto stringSize = std::strlen(cStyleString);
        _isAllocated = (stringSize > SMALL_BUFFER_SIZE);
        if (_isAllocated)
            _buffer = new char[stringSize];
        else
            _buffer = &_smallBuffer[0];
        std::strcpy(_buffer, &cStyleString[0]);
    }

    string(string &&rhs)
       : _isAllocated(rhs._isAllocated)
       , _buffer(rhs._buffer)
       , _smallBuffer(rhs._smallBuffer) //< Not needed if allocated
    {
        if (_isAllocated)
        {
           // Prevent double deletion of the memory
           rhs._buffer = nullptr;
        }
        else
        {
            // Copy over data
            std::strcpy(_smallBuffer, rhs._smallBuffer);
            _buffer = &_smallBuffer[0];
        }
```

```
    }
    // Other methods, including other constructors, copy constructor,
    // assignment operators have been omitted for readability
};
```

As you can see in the code above, some extra complexity has been added in order to prevent some `new` and `delete` operations. On top of this, the class has a larger memory footprint which might not be used except in a couple of cases.

Often it is tried to encode the bool value `_isAllocated`, within the pointer `_buffer` with bit manipulation to reduce the size of a single instance (intel 64 bit: Could reduce size by 8 byte). An optimization which is only possible when its known what the alignment rules of the platform is.

When to use?

As this optimization adds a lot of complexity, it is not recommended to use this optimization on every single class. It will often be encountered in commonly used, low-level data structures. In common C++11 `standard library` implementations one can find usages in `std::basic_string<>` and `std::function<>`.

As this optimization only prevents memory allocations when the stored data is smaller than the buffer, it will only give benefits if the class is often used with small data.

A final drawback of this optimization is that extra effort is required when moving the buffer, making the move-operation more expensive than when the buffer would not be used. This is especially true when the buffer contains a non-POD type.

Chapter 144: Optimization

When compiling, the compiler will often modify the program to increase performance. This is permitted by the as-if rule, which allows any and all transformations that do not change observable behavior.

Section 144.1: Inline Expansion/Inlining

Inline expansion (also known as inlining) is compiler optimisation that replaces a call to a function with the body of that function. This saves the function call overhead, but at the cost of space, since the function may be duplicated several times.

```
// source:

int process(int value)
{
    return 2 * value;
}

int foo(int a)
{
    return process(a);
}

// program, after inlining:

int foo(int a)
{
    return 2 * a; // the body of process() is copied into foo()
}
```

Inlining is most commonly done for small functions, where the function call overhead is significant compared to the size of the function body.

Section 144.2: Empty base optimization

The size of any object or member subobject is required to be at least 1 even if the type is an empty `class` type (that is, a `class` or `struct` that has no non-static data members), in order to be able to guarantee that the addresses of distinct objects of the same type are always distinct.

However, base `class` subobjects are not so constrained, and can be completely optimized out from the object layout:

```
#include <cassert>

struct Base {}; // empty class

struct Derived1 : Base {
    int i;
};

int main() {
    // the size of any object of empty class type is at least 1
    assert(sizeof(Base) == 1);

    // empty base optimization applies
    assert(sizeof(Derived1) == sizeof(int));
```

```
}
```

Empty base optimization is commonly used by allocator-aware standard library classes (`std::vector`, `std::function`, `std::shared_ptr`, etc) to avoid occupying any additional storage for its allocator member if the allocator is stateless. This is achieved by storing one of the required data members (e.g., `begin`, `end`, or `capacity` pointer for the `vector`).

Reference: cppreference

Chapter 145: Profiling

Section 145.1: Profiling with gcc and gprof

The GNU gprof profiler, gprof, allows you to profile your code. To use it, you need to perform the following steps:

1. Build the application with settings for generating profiling information
2. Generate profiling information by running the built application
3. View the generated profiling information with gprof

In order to build the application with settings for generating profiling information, we add the `-pg` flag. So, for example, we could use

```
$ gcc -pg *.cpp -o app
```

or

```
$ gcc -O2 -pg *.cpp -o app
```

and so forth.

Once the application, say `app`, is built, execute it as usual:

```
$ ./app
```

This should produce a file called `gmon.out`.

To see the profiling results, now run

```
$ gprof app gmon.out
```

(note that we provide both the application as well as the generated output).

Of course, you can also pipe or redirect:

```
$ gprof app gmon.out | less
```

and so forth.

The result of the last command should be a table, whose rows are the functions, and whose columns indicate the number of calls, total time spent, self time spent (that is, time spent in the function excluding calls to children).

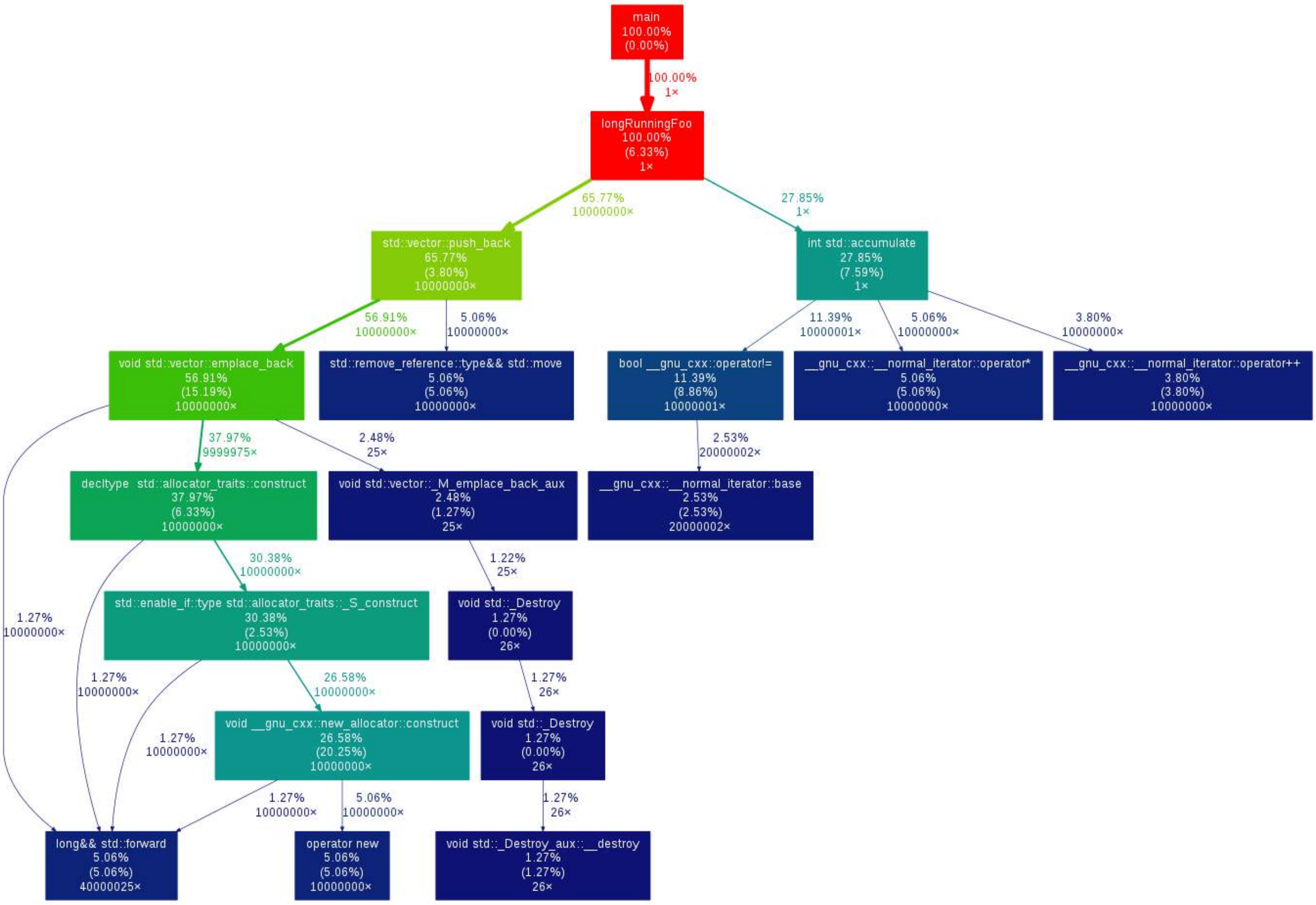
Section 145.2: Generating callgraph diagrams with gperf2dot

For more complex applications, flat execution profiles may be difficult to follow. This is why many profiling tools also generate some form of annotated callgraph information.

gperf2dot converts text output from many profilers (Linux perf, callgrind, oprofile etc.) into a callgraph diagram. You can use it by running your profiler (example for `gprof`):

```
# compile with profiling flags
g++ *.cpp -pg
```

```
# run to generate profiling data
./main
# translate profiling data to text, create image
gprof ./main | gprof2dot -s | dot -Tpng -o output.png
```



Section 145.3: Profiling CPU Usage with gcc and Google Perf Tools

Google Perf Tools also provides a CPU profiler, with a slightly friendlier interface. To use it:

1. Install Google Perf Tools
2. Compile your code as usual
3. Add the `libprofiler` profiler library to your library load path at runtime
4. Use `pprof` to generate a flat execution profile, or a callgraph diagram

For example:

```
# compile code
g++ -O3 -std=c++11 main.cpp -o main

# run with profiler
LD_PRELOAD=/usr/local/lib/libprofiler.so CPUPROFILE=main.prof CPUPROFILE_FREQUENCY=100000 ./main
```

where:

- `CPUPROFILE` indicates the output file for profiling data
- `CPUPROFILE_FREQUENCY` indicates the profiler sampling frequency;

Use `pprof` to post-process the profiling data.

You can generate a flat call profile as text:

```
$ pprof --text ./main main.prof
PROFILE: interrupts/evictions/bytes = 67/15/2016
pprof --text --lines ./main main.prof
Using local file ./main.
Using local file main.prof.
Total: 67 samples
22  32.8%  32.8%       67 100.0% longRunningFoo ??:0
20  29.9%  62.7%       20  29.9% __memmove_ssse3_back
/build/eglibc-3GlaMS/eglibc-2.19/string/../sysdeps/x86_64/multiarch/memcpy-ssse3-back.S:1627
4   6.0%  68.7%        4   6.0% __memmove_ssse3_back
/build/eglibc-3GlaMS/eglibc-2.19/string/../sysdeps/x86_64/multiarch/memcpy-ssse3-back.S:1619
3   4.5%  73.1%        3   4.5% __random_r /build/eglibc-3GlaMS/eglibc-2.19/stdlib/random_r.c:388
3   4.5%  77.6%        3   4.5% __random_r /build/eglibc-3GlaMS/eglibc-2.19/stdlib/random_r.c:401
2   3.0%  80.6%        2   3.0% __munmap
/build/eglibc-3GlaMS/eglibc-2.19/misc/../sysdeps/unix/syscall-template.S:81
2   3.0%  83.6%       12  17.9% __random /build/eglibc-3GlaMS/eglibc-2.19/stdlib/random.c:298
2   3.0%  86.6%        2   3.0% __random_r /build/eglibc-3GlaMS/eglibc-2.19/stdlib/random_r.c:385
2   3.0%  89.6%        2   3.0% rand /build/eglibc-3GlaMS/eglibc-2.19/stdlib/rand.c:26
1   1.5%  91.0%        1   1.5% __memmove_ssse3_back
/build/eglibc-3GlaMS/eglibc-2.19/string/../sysdeps/x86_64/multiarch/memcpy-ssse3-back.S:1617
1   1.5%  92.5%        1   1.5% __memmove_ssse3_back
/build/eglibc-3GlaMS/eglibc-2.19/string/../sysdeps/x86_64/multiarch/memcpy-ssse3-back.S:1623
1   1.5%  94.0%        1   1.5% __random /build/eglibc-3GlaMS/eglibc-2.19/stdlib/random.c:293
1   1.5%  95.5%        1   1.5% __random /build/eglibc-3GlaMS/eglibc-2.19/stdlib/random.c:296
1   1.5%  97.0%        1   1.5% __random_r /build/eglibc-3GlaMS/eglibc-2.19/stdlib/random_r.c:371
1   1.5%  98.5%        1   1.5% __random_r /build/eglibc-3GlaMS/eglibc-2.19/stdlib/random_r.c:381
1   1.5% 100.0%        1   1.5% rand /build/eglibc-3GlaMS/eglibc-2.19/stdlib/rand.c:28
0   0.0% 100.0%       67 100.0% __libc_start_main /build/eglibc-3GlaMS/eglibc-2.19/csu/libc-
start.c:287
0   0.0% 100.0%       67 100.0% _start ??:0
0   0.0% 100.0%       67 100.0% main ??:0
0   0.0% 100.0%       14  20.9% rand /build/eglibc-3GlaMS/eglibc-2.19/stdlib/rand.c:27
0   0.0% 100.0%       27  40.3% std::vector::_M_emplace_back_aux ??:0
```

... or you can generate an annotated callgraph in a pdf with:

```
pprof --pdf ./main main.prof > out.pdf
```

Chapter 146: Refactoring Techniques

Refactoring refers to the modification of existing code into an improved version. Although refactoring is often done while changing code to add features or fix bugs, the term particularly refers improving code without necessarily adding features or fixing bugs.

Section 146.1: Goto Cleanup

In C++ code bases which used to be C, one can find the pattern `goto cleanup`. As the `goto` command makes the workflow of a function harder to understand, this is often avoided. Often, it can be replaced by return statements, loops, functions. Though, with the `goto cleanup` one needs to get rid of cleanup logic.

```
short calculate(VectorStr **data) {
    short result = FALSE;
    VectorStr *vec = NULL;
    if (!data)
       goto cleanup;  //< Could become return false

    // ... Calculation which 'new's VectorStr

    result = TRUE;
cleanup:
    delete [] vec;
    return result;
}
```

In C++ one could use RAII to fix this issue:

```
struct VectorRAII final {
    VectorStr *data{nullptr};
    VectorRAII() = default;
    ~VectorRAII() {
        delete [] data;
    }
    VectorRAII(const VectorRAII &) = delete;
};

short calculate(VectorStr **data) {
    VectorRAII vec{};
    if (!data)
       return FALSE;  //< Could become return false

    // ... Calculation which 'new's VectorStr and stores it in vec.data

    return TRUE;
}
```

From this point on, one could continue refactoring the actual code. For example by replacing the `VectorRAII` by `std::unique_ptr` or `std::vector`.

Credits

Thank you greatly to all the people from Stack Overflow Documentation who helped provide this content, more changes can be sent to web@petercv.com for new content to be published or updated

0x5f3759df	Chapter 38
1337ninja	Chapter 47
3442	Chapter 47
4444	Chapter 143
A. Sarid	Chapters 6 and 25
aaronsnoswell	Chapter 24
Abhinav Gauniyal	Chapter 127
Abyx	Chapter 33
Adam Trhon	Chapter 142
Adhokshaj Mishra	Chapter 138
Aditya	Chapter 23
Ajay	Chapters 7, 33, 73, 102 and 119
alain	Chapter 73
Alejandro	Chapter 80
Alexey Guseynov	Chapter 72
Alexey Voytenko	Chapters 33 and 34
alter igel	Chapter 35
amanuel2	Chapters 21, 29, 32, 39 and 128
amchacon	Chapter 80
Ami Tavory	Chapters 13, 49, 62, 104, 130 and 145
an0o0nym	Chapter 3
anatolyg	Chapters 49 and 67
anderas	Chapters 12, 16, 33, 34, 44, 49 and 73
Andrea Chua	Chapters 44, 96 and 131
Andrea Corbelli	Chapters 26, 47, 50 and 73
AndyG	Chapters 49 and 110
Anonymous1847	Chapter 132
anotherGatsby	Chapters 11 and 15
Antonio Barreto	Chapter 112
AProgrammer	Chapter 12
Aravind .KEN	Chapter 39
ArchbishopOfBanterbury	Chapters 1, 36 and 138
Artalus	Chapter 108
asantacreu	Chapter 146
Asu	Chapter 26
Ates Goral	Chapter 36
Bakhtiar Hasan	Chapter 35
Baron Akramovic	Chapter 30
Barry	Chapters 6, 9, 11, 16, 18, 24, 33, 36, 40, 44, 47, 49, 51, 63, 67, 73, 74, 77, 79, 82, 83, 98, 103, 105, 108, 110 and 138
bcmpinc	Chapter 73
Ben H	Chapter 1
Ben Steffan	Chapter 138
Benjy Kessler	Chapter 77
BigONotation	Chapter 78
Bim	Chapters 39 and 54

Brian	Chapters 1, 2, 3, 15, 20, 21, 22, 23, 34, 36, 44, 46, 64, 69, 71, 72, 73, 77, 79, 80, 84, 91, 95, 97, 98, 99, 100, 104, 105, 115, 119, 120, 121, 133 and 134
C.W.Holeman II	Chapter 63
CaffeineToCode	Chapters 33 and 80
callyalater	Chapters 34, 72 and 75
Candlemancer	Chapter 36
caps	Chapter 47
cb4	Chapter 77
celtschk	Chapters 1, 16, 24, 77, 90, 108 and 124
Chachmu	Chapter 12
Cheers and hth.	Chapters 1, 8, 75, 94 and 106
chema989	Chapter 144
ChemiCalChems	Chapters 11, 74 and 106
CHess	Chapter 49
chrisb2244	Chapters 1 and 34
ChrisN	Chapter 11
Christophe	Chapter 25
Christopher Oezbek	Chapters 24, 33, 47 and 73
Cid1025	Chapter 114
CinCout	Chapters 48 and 49
CodeMouse92	Chapter 77
Cody Gray	Chapters 1, 6 and 49
CoffeeandCode	Chapter 35
Colin Basnett	Chapters 16, 34, 49 and 77
ColleenV	Chapter 11
ComicSansMS	Chapters 12, 33, 49 and 80
cpplearner	Chapter 33
crea7or	Chapter 47
CroCo	Chapter 6
cshu	Chapter 104
Curious	Chapters 1 and 47
cute_ptr	Chapters 9 and 49
CygnusX1	Chapters 50 and 75
Daemon	Chapters 1 and 10
Daksh Gupta	Chapters 1, 26, 33, 38, 48, 49 and 115
Dan Hulme	Chapter 34
Danh	Chapter 107
Daniel	Chapters 62 and 67
Daniel Jour	Chapter 9
Daniel Käfer	Chapter 69
Daniel Stradowski	Chapter 49
Daniele Pallastrelli	Chapters 33, 107 and 108
darkpsychic	Chapters 1 and 26
davidsheldon	Chapter 50
DawidPi	Chapter 16
Dean Seo	Chapter 18
DeepCoder	Chapters 1, 24, 49 and 77
deepmax	Chapters 16, 84, 113 and 117
define cindy const	Chapter 99
defube	Chapters 36, 80 and 83
demonplus	Chapters 54 and 58
Denkkar	Chapter 68
didiz	Chapters 3, 80, 81, 85, 86, 87, 88, 105, 112 and 140

diegodfrf	Chapters 49, 119 and 135
Dietmar Kühl	Chapter 60
Dim_ov	Chapter 1
dkg	Chapter 49
Donald Duck	Chapters 1 and 138
Dr t	Chapters 1, 12, 49 and 72
Dragma	Chapter 34
drov	Chapter 47
Duly Kinsky	Chapters 49 and 62
Dutow	Chapter 71
Edd	Chapter 73
Edgar Rokjān	Chapter 62
Edward	Chapters 1, 11, 33, 47, 66, 108 and 146
ehudt	Chapter 49
Ela782	Chapter 24
elimad	Chapter 52
elvis.dukaj	Chapter 141
Emil Rowland	Chapter 47
emlai	Chapter 33
Emmanuel Mathi	Chapters 69 and 98
Enamul Hassan	Chapters 1, 24, 47, 49, 50, 67 and 138
enzom83	Chapter 36
Error	Chapters 48 and 117
Evgeniy	Chapter 52
EvgeniyZh	Chapter 9
Falias	Chapter 49
Fantastic Mr Fox	Chapters 1, 24, 34, 47, 49, 50, 68, 75 and 138
fbrereto	Chapters 9 and 47
FedeWar	Chapters 30 and 77
Florian	Chapters 1 and 130
Fox	Chapters 49 and 75
foxcub	Chapters 33, 49 and 50
Gabriel	Chapter 79
Gal Dreiman	Chapter 9
Galik	Chapters 12, 24, 49, 50, 81 and 113
Gaurav Kumar Garg	Chapter 9
Gaurav Sehgal	Chapter 76
ggrr	Chapter 104
GIRISH kuniyal	Chapter 104
granmirupa	Chapter 49
Greg	Chapter 77
Guillaume Pascal	Chapters 62 and 63
Guillaume Racicot	Chapter 106
Ha.	Chapter 65
hello	Chapter 82
Henkersmann	Chapter 28
Hindrik Stegenga	Chapter 30
holmicz	Chapter 11
Holt	Chapters 16, 47, 49 and 77
honk	Chapters 1, 9, 11, 12, 24, 33, 47, 50, 73, 77, 79 and 82
Humam Helfawi	Chapter 1
Hurkyl	Chapters 9, 12 and 49
hyoslee	Chapter 85

Ian Ringrose	Chapter 75
Igor Oks	Chapter 108
immerhart	Chapters 49 and 139
In silico	Chapter 101
Ivan Kush	Chapter 63
Jérémy Roy	Chapter 44
Jack	Chapter 47
Jahid	Chapters 47, 71, 72 and 138
James Adkison	Chapters 36 and 80
Jared Payne	Chapters 33 and 51
Jarod42	Chapters 6, 16, 17, 25, 34, 44, 68, 72, 78, 83, 90, 103, 108, 112, 113, 114, 120, 121 and 138
Jason Watkins	Chapters 49, 80, 130 and 138
Jatin	Chapters 17 and 35
Jean	Chapter 73
Jerry Coffin	Chapter 34
Jim Clark	Chapter 1
Johan Lundberg	Chapters 1, 24, 33, 49, 82, 108, 113 and 138
Johannes Schaub	Chapters 11, 24, 35, 37, 44, 73, 74, 101, 105 and 122
John Burger	Chapter 31
John DiFini	Chapter 43
John London	Chapter 23
Jonathan Lee	Chapters 78 and 103
Jonathan Mee	Chapters 47 and 70
jotik	Chapters 1, 33, 34, 47, 71, 72, 92 and 138
JPNotADragon	Chapter 9
jpo38	Chapters 47 and 49
Julien	Chapter 44
Justin	Chapters 1, 16, 17, 33, 70, 77 and 130
Justin Time	Chapters 1, 11, 20, 23, 25, 32, 34, 35, 38, 41, 71, 75, 82, 95, 106, 128 and 138
JVApen	Chapters 1, 3, 6, 12, 15, 27, 33, 35, 44, 49, 59, 63, 73, 83, 89, 91, 106, 107, 115, 118, 123, 126, 130, 138, 142, 143 and 146
K48	Chapter 1
kd1508	Chapter 104
Ken Y	Chapters 47 and 75
Kerrek SB	Chapters 21, 33 and 34
Keshav Sharma	Chapter 1
kiner_shah	Chapters 1 and 57
krOoze	Chapters 1, 40, 49 and 75
Kunal Tyagi	Chapter 37
L.V.Rao	Chapter 11
Leandros	Chapter 1
legends2k	Chapter 39
Let_Me_Be	Chapter 24
lorro	Chapters 118 and 143
Loufylouf	Chapter 73
Luc Danton	Chapter 103
maccard	Chapter 67
madduci	Chapters 115 and 138
Malcolm	Chapters 1 and 48
Malick	Chapters 1 and 138
manlio	Chapters 1, 5, 6, 11, 12, 25, 47, 49, 50, 63, 65, 71, 104, 111 and 138
Marc.2377	Chapter 47

marcinj	Chapter 66
Marco A.	Chapters 58, 63, 75, 100, 118 and 129
Mark Gardner	Chapter 1
marquesm91	Chapter 69
Martin York	Chapters 5, 12, 24, 49, 73, 77, 82 and 83
MasterHD	Chapter 1
MathSquared	Chapter 123
Matt	Chapters 1 and 138
Matthew Brien	Chapter 8
Matthieu M.	Chapter 16
Maxito	Chapter 77
Meena Alfons	Chapter 125
merlinND	Chapter 65
Meysam	Chapters 33, 47 and 50
Michael Gaskill	Chapter 138
Mike H	Chapter 9
MikeMB	Chapter 67
Mikitori	Chapter 59
Mimouni	Chapter 1
mindriot	Chapters 12 and 143
Misgevolution	Chapter 142
MKAROL	Chapter 67
mkluwe	Chapter 15
MotKohn	Chapter 49
Motti	Chapters 38, 49 and 104
mpromonet	Chapters 13 and 47
MSalters	Chapters 63 and 77
MSD	Chapter 138
mtb	Chapter 119
mtk	Chapter 49
Muhammad Aladdin	Chapter 1
muXXmit2X	Chapter 1
n.m.	Chapters 75 and 138
Naor Hadar	Chapter 66
Nathan Osman	Chapters 1, 130 and 138
Naveen Mittal	Chapter 50
Neil A.	Chapter 1
Nemanja Boric	Chapters 1, 72 and 138
Niall	Chapters 24, 47, 49 and 83
Nicholas	Chapter 11
Nicol Bolas	Chapters 11, 12, 16, 24, 27, 30, 44, 49, 52, 67, 68, 73, 74, 75, 79, 82, 88, 100 and 109
Nikola Vasilev	Chapters 2, 48, 53, 54, 55, 57, 91 and 112
Nitinkumar Ambekar	Chapter 30
nnrales	Chapter 115
NonNumeric	Chapter 116
Null	Chapters 11, 44, 47, 50, 72 and 82
nwp	Chapter 80
Omnifarious	Chapter 20
Oz.	Chapter 9
pandaman1234	Chapter 49
Pankaj Kumar Boora	Chapter 84
patmanpato	Chapters 12 and 49

Patryk Obara	Chapter 62
paul	Chapters 49 and 145
Paul Beckingham	Chapter 49
Pavel Strakhov	Chapter 1
PcAF	Chapters 33 and 34
Ped7g	Chapters 11 and 49
Perette Barella	Chapter 7
Peter	Chapters 24, 50, 71, 75, 104 and 138
phandinhlan	Chapter 75
Pietro Saccardi	Chapter 30
plasmacel	Chapter 48
pmelanson	Chapter 11
Podgorskiy	Chapter 17
Praetorian	Chapters 49 and 73
Pyves	Chapter 11
Qchmqs	Chapter 15
Quirk	Chapters 1 and 138
R. Martinho Fernandes	Chapter 49
Rakete1111	Chapters 11, 12, 24, 33, 34, 35, 36, 44, 47, 49, 73, 77, 80, 104 and 110
ralismark	Chapters 104 and 144
RamenChef	Chapters 2, 15, 20, 21, 22 and 69
Ravi Chandra	Chapters 54 and 67
Reuben Thomas	Chapter 40
Richard Dally	Chapters 33, 47, 49, 50, 75 and 138
rockoder	Chapter 26
rodrigo	Chapter 33
Roland	Chapters 33, 44, 84, 92, 101 and 114
RomCoo	Chapter 12
Ronen Ness	Chapter 72
rtmh	Chapter 16
Rushikesh Deshpande	Chapters 1 and 49
Ryan Haining	Chapters 73 and 79
R_Kapp	Chapter 124
Saint	Chapter 49
SajithP	Chapter 67
Samer Tufail	Chapter 49
Sean	Chapters 35 and 75
Sergey	Chapters 9, 13, 19, 34, 38, 77 and 138
Serikov	Chapters 12, 47, 49 and 73
Shoe	Chapters 1 and 49
sigalor	Chapter 114
silvergasp	Chapters 34, 49, 75 and 93
SingerOfTheFall	Chapter 123
SirGuy	Chapter 1
Skipper	Chapters 47 and 49
Skywrath	Chapter 34
Smeeheey	Chapter 77
Snowhawk	Chapter 73
SouvikMaji	Chapter 50
sp2danny	Chapter 103
stackptr	Chapter 68
start2learn	Chapters 36 and 133
Stephen	Chapters 24, 49 and 89

sth	Chapters 16 and 49
Stradigos	Chapter 113
strangeqargo	Chapter 49
SU3	Chapter 103
Sumurai8	Chapters 33, 65 and 83
T.C.	Chapters 38 and 49
tambre	Chapter 6
Tannin	Chapters 83 and 104
Tarod	Chapter 52
TartanLlama	Chapters 93 and 109
Tejendra	Chapter 15
tenpercent	Chapters 17, 18, 24, 44 and 75
Tharindu Kumara	Chapter 138
The Philomath	Chapter 75
theo2003	Chapter 49
ThyReaper	Chapters 17, 92, 111 and 115
Toby	Chapter 138
Tom	Chapter 49
towi	Chapter 49
Trevor Hickey	Chapters 6, 67 and 104
TriskalJM	Chapters 1, 40, 49 and 82
Trygve Laugstøl	Chapter 138
tulak.hord	Chapter 61
turoni	Chapter 3
txtechhelp	Chapters 5 and 111
UncleZeiv	Chapter 1
user1336087	Chapters 47, 49 and 50
user2176127	Chapters 47 and 49
user3384414	Chapter 24
user3684240	Chapter 33
vdaras	Chapter 50
Venki	Chapter 16
VermillionAzure	Chapters 1, 45, 130 and 141
Vijayabhaskarreddy CH	Chapter 42
Ville	Chapters 1 and 24
Vladimir Gamalyan	Chapter 49
VladimirS	Chapter 11
VolkA	Chapter 50
W.F.	Chapters 16 and 77
w1th0utnam3	Chapter 103
Walter	Chapter 1
wasthishelpful	Chapter 137
Wolf	Chapters 8, 47, 49 and 77
WQYeo	Chapters 1 and 8
Wyzard	Chapter 50
Xirema	Chapter 4
Yakk	Chapters 9, 11, 24, 33, 36, 42, 44, 49, 51, 56, 57, 73, 80, 90, 103, 107, 108 and 121
Yousuf Azad	Chapter 33
ysdx	Chapter 16
Yuushi	Chapter 80
ΦXocę 웃 Пepeúpa ッ	Chapter 8
Алексей Неудачин	Chapters 5 and 12

Владимир Стрелец	Chapters 10 and 14
パスカル	Chapters 1, 75 and 136